Debugging Memory Problems using TotalView

ETNUS

November 2004

version 6.6

Contents

1 Debugging Memory Problems

Any time you read about debugging, you read that 60 or 70% of all programming errors are memory-related. So, while these numbers may be wrong, let's assume that they are right. Now for the bad news: the reason that memory errors occur is that the programmer made an error. All memory errors are preventable.

Why are there so many memory errors? There are many answers. For example, programs are complicated. And, programmers make assumptions when they shouldn't. Is a library function allocating its own memory or should the program be allocating it? Once it is allocated, does your program manage the memory or does the library? Something creates a pointer to something and the memory is freed without any knowledge that something else is pointing to it. Or, and these are the most prevalent reason, there's a wide separation between lines of code or the time when old code and new code was written. And, of course, there's always insufficient and bad documentation.

Some problems can be irrelevant. If you forget to free the memory allocated for a small array, it doesn't mean much. And, it may even be more efficient not to free the memory. The operating system will free it for you when the program ends, so there are times when you don't want to bother. On the other hand, if you continually allocate memory without freeing it, your program may eventually crash because it can't get more memory.

Checking for Problems

The TotalView Memory Debugger can help you locate many of your program's memory problems. For example, you can:

- Stop execution when **free()**, **realloc()**, and other heap API problems occur.

  If your program tries to free memory that it can't or shouldn't free, the Memory Debugger can stop execution. This lets you identify the statement that caused the problem. For more information, see "*Finding free() and realloc() Problems*" on page 17.
- List leaks.

  The Memory Debugger can display your program's leaks. (*Leaks* are memory blocks that are allocated, but which are no longer referenced.)

  When your program allocates a memory block, the Memory Debugger creates a backtrace. When it makes a list of your leaks, it includes this backtrace in the list. This lets you see the place where your program allocated the memory block. For more information, see "*Finding Memory Leaks*" on page 21.
- Paint allocated and deallocated blocks.

  When your program's memory manager allocates or deallocates memory, the Memory Debugger can write a bit pattern into it. Writing this bit pattern is called *painting*.

  When you see this bit pattern in a Variable or Expression List Window, you know that you are using memory before your program initializes it or after your program deallocates it. Depending upon the architecture, you might even be able to force an exception when your program accesses this memory. For more information, see "*Block Painting*" on page 26.
- Identify dangling pointers.

  A *dangling pointer* is a pointer that points into deallocated memory. If the pointer being displayed in a Variable Window is dangling, TotalView adds information to the data element so that you know about the problem. For more information, see "*Dangling Pointers*" on page 24.
- Hold onto deallocated memory.

  When trying to identify memory problems, holding onto memory after your program releases it can sometimes help locate problems by forcing a memory error to occur. Holding onto freed memory is called *hoarding*.

  If you are also painting memory, you can know when your program is trying to access deallocated memory. For more information, see "*Hoarding*" on page 27.

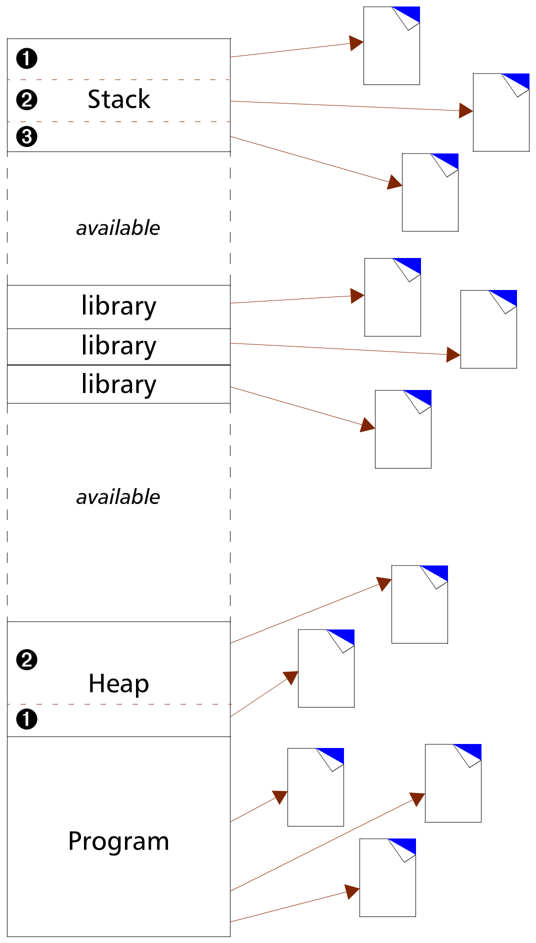
Programs and Memory

When you run a program, your operating system loads the program into memory and defines an address space in which the program can operate. For example, if your program is executing in a 32-bit computer, the address space is approximately 4 gigabytes.

Since the discussion in this chapter is pretty general, what you will be reading is almost true for many computer architectures, somewhat wrong for all, and perhaps completely wrong for the computer upon which you are debugging memory problems. For accurate information, you'll need to read information provided by your vendor.

The operating system does not actually allocate the memory in this address space. Instead, operating systems memory map this space, which means that it maps the relationship between the theoretical address space your program could use and what it actually uses. Typically, operating systems divide memory into pages. When a program begins executing, the operating system creates a map that correlates the executing program with the pages that contain the program's information. The following figure shows regions of a program. The arrows point to the memory pages that contain the program.

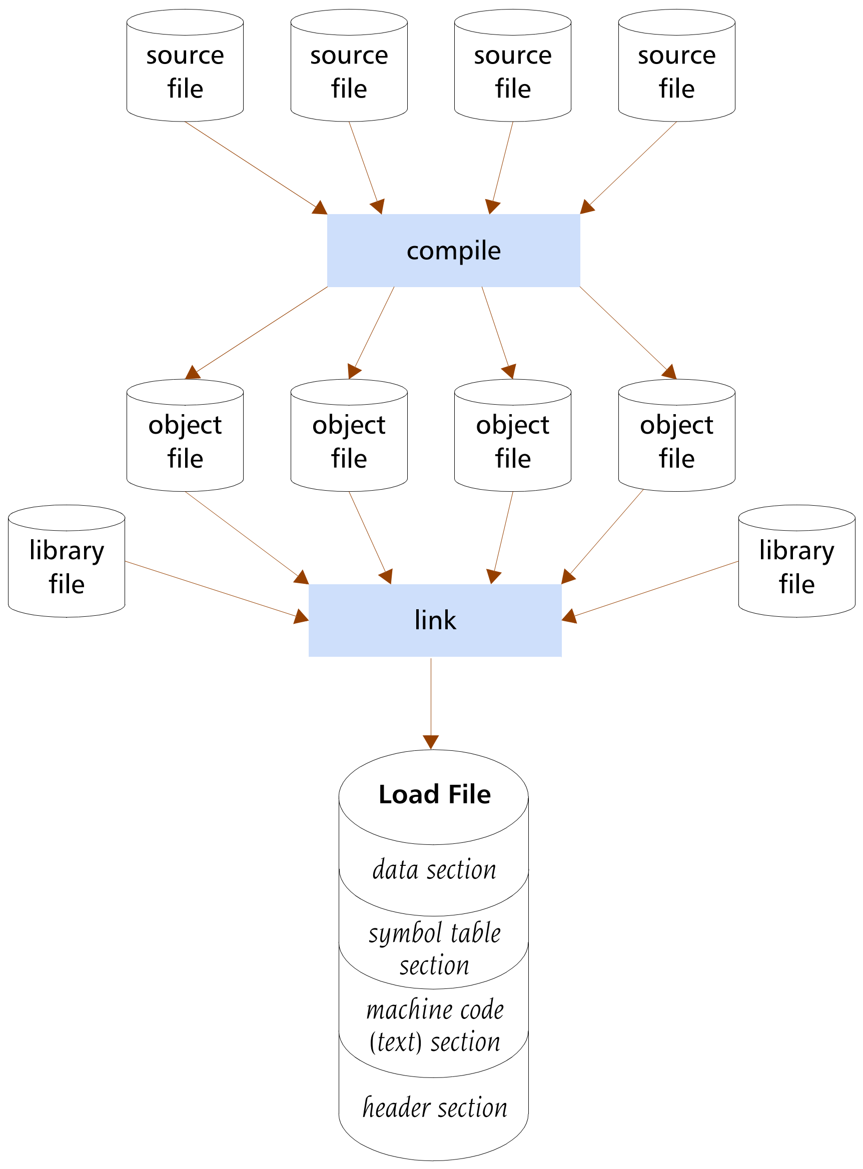
Figure 1: Mapping Program Pages



In this figure, the stack contains three stack frames, each mapped to its own page. Similarly, the heap shows two allocations, each of which is mapped to its own page. (This isn't what really happens since a page can have many stack frames and many heap allocations. But doing this makes a nice picture.)

The program did not emerge fully-formed into this state. It had to be compiled, linked, and loaded. The following figure shows a program whose source code resides in four files. Running these files through a compiler creates object files. A linker then merges these object files and any external libraries needed into a load file. This load file is the executable program that is stored on your computer's file system.

Figure 2: Compiling Programs



When the linker creates the load file, it combines the information contained in each of the object files into one unit. Combining them is relatively straightforward. The load file shown at the bottom of this figure simplifies this file's contents, since it always contains more sections and more information.

The contents of these sections are as follows:

- **Data section**—contains static variables and variables initialized outside of a function. The following is a small sample program:

  ```
  int my_var1 = 10;
  void main ()
  {
     static int my_var2 = 1;
     int my_var3;
     my_var3 = my_var1 + my_var2;
     printf(“here’s what I’ve got: %i\n”, my_var3);
  }
  ```

  The data section contains the **my_var1** and **my_var2** variables. The memory for the **my_var3** variable is dynamically and automatically allocated within the stack by your program’s runtime system.
- **Symbol table section**—contains addresses (usually offsets) to the locations of routines and variables.
- **Machine code section**—contains an intermediate binary representation of your program. (It is intermediate because addresses are not yet resolved.)
- **Header section**—contains information about the size and location of information in all other sections of the object file.

When the linker creates the load file from the object and library files, it interweaves these sections into one file. The linking operation creates something that your operating system can load into memory. Figure 3 on page 6 shows this process.

The Memory Debugger can provide information about these sections and the amount of memory your program is using. To obtain this information, select the **Tools** > **Memory Debugging** command and then select the **Memory Usage** tab and select Process View. (See Figure 4 on page 7.)

In this listing, the data and symbol table sections of the load file are combined into the **Data** column.

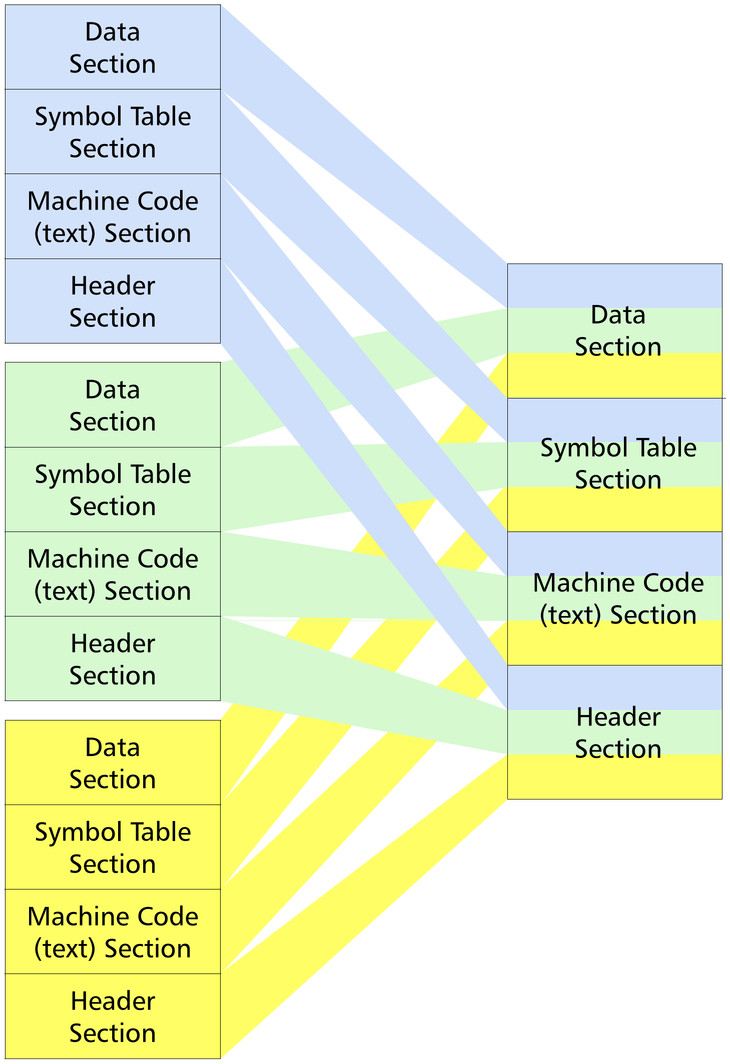
For information on this page, see “*Memory Usage Page*” on page 42.

Behind the Scenes

The TotalView Memory Debugger intercepts calls made by your program to heap library functions that allocate and deallocate memory using the **malloc()** and **free()** functions and the **new** and **delete** operators. It also tracks related functions such, as **calloc()** and **realloc()**. The Memory Debugger uses a technique called interposition, in which an agent intercepts calls to functions.

You can use the Memory Debugger with any allocation and deallocation library that uses such functions as **malloc()** and **free()**. For example, the C++ **new** operator is almost always built on top of the **malloc()** function. If it is, the Memory Debugger can track it. Similarly, some Fortran implementations use the **malloc()** and **free()** functions to manage memory. In these cases, the Memory Debugger can track Fortran memory use.

Figure 3: Linking a Program



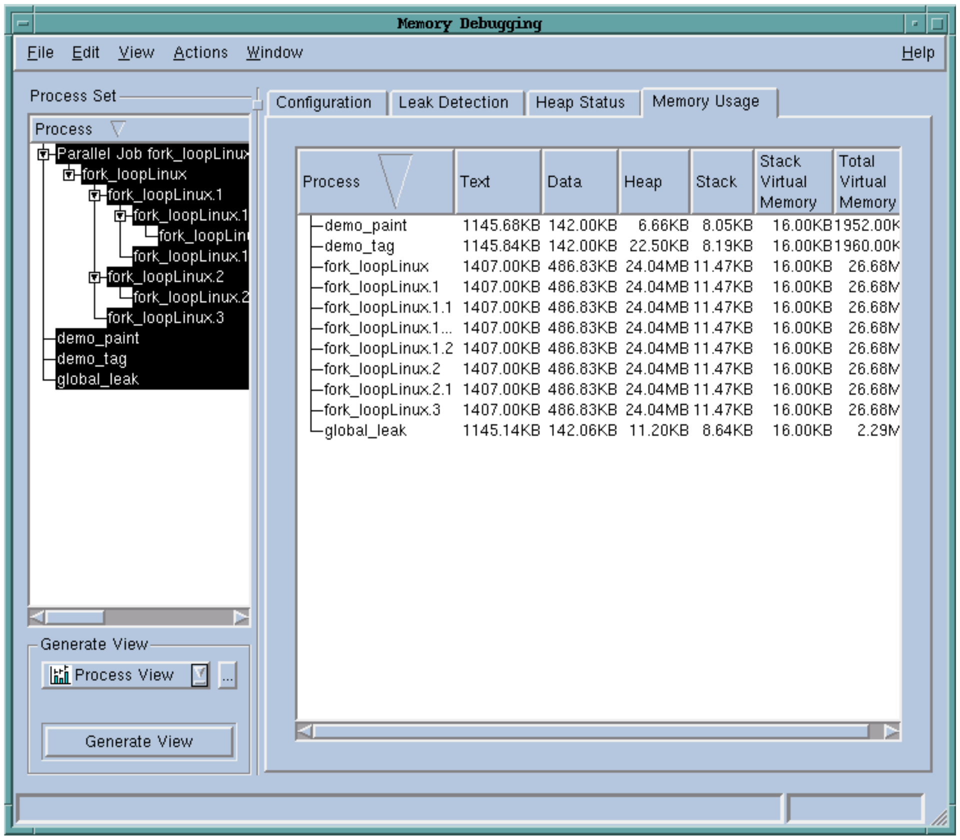
You can interpose the agent in two ways:

- You can tell TotalView to preload the agent. *Preloading* means that the loader loads an object before the object listed in the application's loader table.

  When a routine references a symbol in another routine, the linker searches for the first definition of that symbol. Because the agent's routine is the first object in the table, its routine is invoked instead of the routine in the program's heap manager.

  On Linux, HP Tru64 Alpha, Sun, and SGI, TotalView sets an environment variable that contains the pathname of the agent's shared library in your local TotalView installation. For more information, see "*Attaching to Programs*" on page 63.
- If TotalView cannot preload the agent, you must explicitly link it into your program. For details, see "*Creating Programs for Memory Debugging*" on page 61.

  If your program attaches to an already running program, you must explicitly link this other program with the agent.



Process	Text	Data	Heap	Stack	Stack Virtual Memory	Total Virtual Memory
demo_paint	1145.68KB	142.00KB	6.66KB	8.05KB	16.00KB	1952.00K
demo_tag	1145.84KB	142.00KB	22.50KB	8.19KB	16.00KB	1960.00K
fork_loopLinux	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1...	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1.2	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.2	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.2.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.3	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
global_leak	1145.14KB	142.06KB	11.20KB	8.64KB	16.00KB	2.29M

Figure 4: Memory Usage Page: Process View

The agent uses operations defined in the dynamic linker's API to find the original definition of the routine. After the agent intercepts a call, it calls the original function. This means that you can use the Memory Debugger with most memory allocators. The following figure shows how the agent interacts with your program and the heap library.

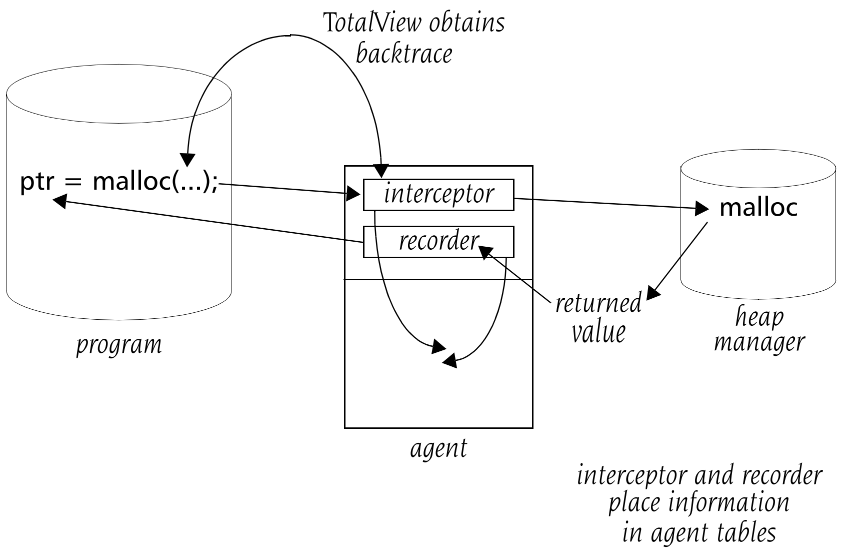

TotalView obtains backtrace
ptr = malloc(...);
program
interceptor
recorder
agent
malloc
heap manager
returned value
interceptor and recorder place information in agent tables

Figure 5: Interposition

Because TotalView uses interposition, memory debugging can be considered non-invasive. That is, TotalView doesn't rewrite or augment your program's code, and you don't have to do anything in your program. Adding the agent does not change your program's behavior.

Your Program's Data

Your program's variables resides in the following places:

- Data section
- Stack
- Heap

The Data Section

Memory in the data section is permanently allocated. Your program uses this section for storing static and global variables. The size of this section is fixed when the operating system loads the program and the variables within it exist for the entire time that your program is executing. Errors can occur if your program tries to manage this section's memory. For example, you cannot free memory allocated to variables in the data section. In general, errors are usually related to the programmer not understanding that the program can't manage data section memory.

The Stack

Memory in the stack section is dynamically managed by your program's memory manager. Consequently, your program cannot allocate memory within the stack or deallocate memory within it.

"Deallocates means that your program is no longer using this memory. The next time your program calls a routine, the new stack frame overwrites the memory previously used by other routines. In almost all cases, deallocated memory, whether on the stack or the heap, just hangs around in its preallocation state until it gets reassigned.

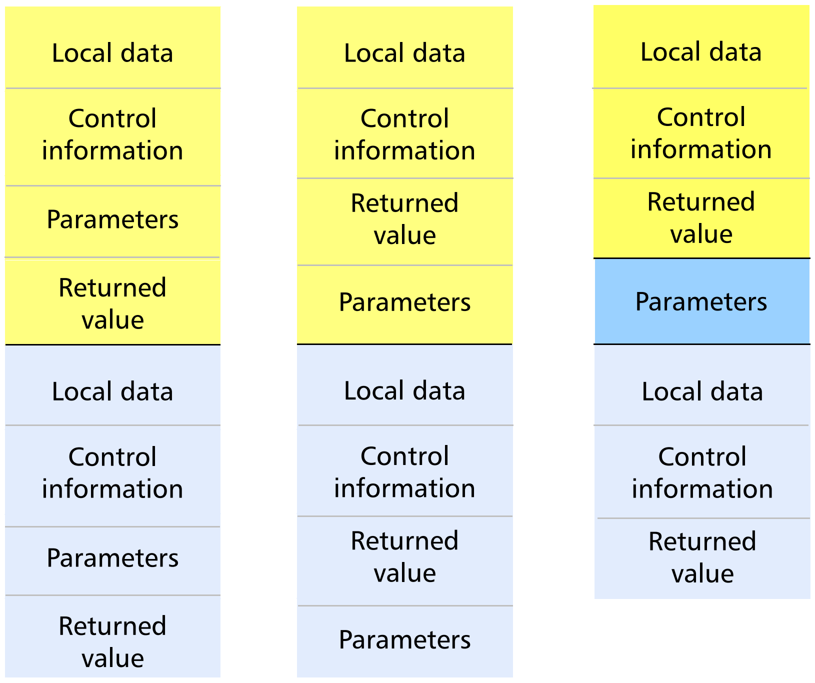
The stack differs from the data section in that the space is dynamically managed. What's in it one minute might not be there a moment later. Your program's runtime environment allocates memory for stack frames as your program calls routines and deallocates these frames when execution exits from it.

At a minimum, a stack frame contains lots of control information, data storage, and space for passed-in arguments (parameters) and the returned value. Figure 6 on page 9 shows three ways in which a compiler can arrange stack frame information:

In this figure, the left and center stack frames have different positions for the parameters and returned value. The stack frame on the right is a little more complicated. In this version, the parameters are located within a stack memory area that doesn't belong to either stack frame.

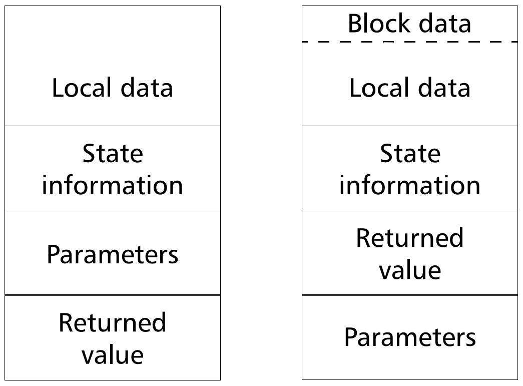
If a stack frame contains local (sometimes called automatic) variables, where is this memory placed? If the routine has blocks in which memory is allocated, where on the stack is this memory for these additional variables

Figure 6: Placing Parameters



placed? Although there are many variations, the following figure shows two of the more common ways to allocate memory:

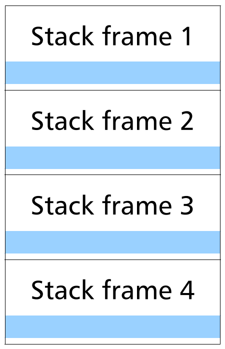
Figure 7: Local Data in a Stack Frame



The blocks on the left shows a data block allocated within a stack frame on a system that ignores your routine's block structure. The compiler figures how much memory is needed, and then allocates enough memory for all of your routine's automatic variables. These kinds of systems are optimized to minimize the time necessary to allocate memory. Other systems dynamically allocate the memory required for a block as the block is entered, and then deallocate it as execution leaves the block. (The blocks on the right show this.) These kinds of systems are optimized to minimize a routine's size.

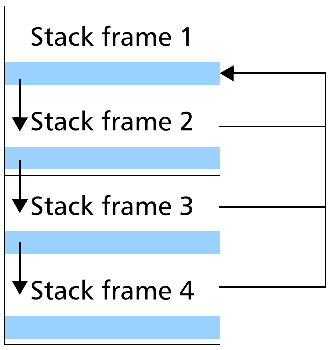
As your program executes routines, routines call other routines, placing additional routines on the stack. The following figure shows four stack frames. The shaded areas represents local data.

Figure 8: Four Stack Frames



What happens when a pointer to memory in a stack frame is passed to lower frames? This situation is shown in the following figure:

Figure 9: Passing Pointers



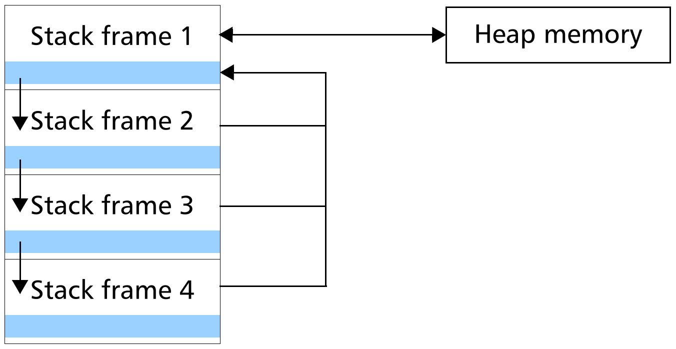
The arrows on the left represent the pointer passed down the stack. The lines and arrows on the right indicate the place to which the pointer is pointing. A pointer to memory in frame 1 is passed to frame 2, which passes the pointer to frame 3, and then to frame 4. In all frames, the pointer points to a memory location in frame 1. Stated in another way, the pointers in frames 2, 3, and 4 point to memory in another stack frame. This is considered the most efficient way for your program to pass data from one routine to another. Using the pointer, you can both access and alter the information that the pointer is pointing to.

Sometimes you read that data can be passed by-value (which means copying it) or by-reference (which means passing a pointer). This really isn't true. Something is always copied. "Pass-by-reference" means that instead of copying the data, the program copies a pointer to the data.

Because the program's run-time system owns stack memory, you cannot free it. Instead, it gets freed when a frame is popped from the stack.

One of the reasons for memory problems is that you it may sometimes be unclear who owns a variable's memory. For example, in the following figure, the routine in frame 1 has allocated memory in the heap, and passes a pointer to that memory to other stack frames:

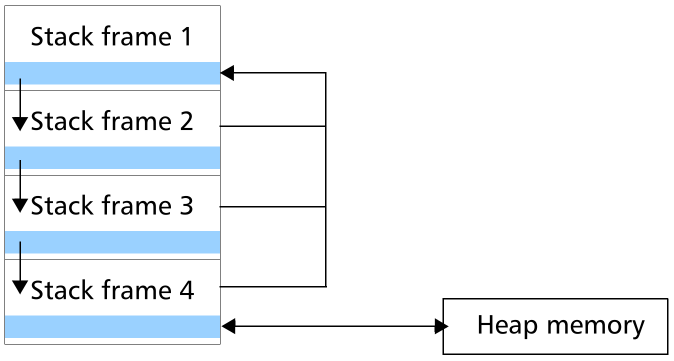
Figure 10: Allocating a Memory Block



If the routine executing in frame 4 frees this memory, all pointers to that memory are dangling; that is, they point to deallocated memory. If the program's memory manager reallocates this heap memory block, the data accessible by all the pointers is both invalid and wrong. Unfortunately, if the memory manager doesn't immediately reuse the block, the data accessed through the pointers is still correct. This is unfortunate, because there's no guarantee that the data is correct and there won't be any pattern to when the block becomes invalid. This means that when problems occur, they are intermittent, which makes them even harder to locate.

Another common problem is when you allocate memory and assign its location to an automatic variable. This is shown in the following figure:

Figure 11: Allocating a Block form a Stack Frame



If frame 4 returns control to frame 3 without deallocating the heap memory it created, this memory is no longer accessible. That is, your program loses the ability to use this memory block. It has *leaked* this memory block.

If you have trouble remembering the difference between a leak and a dangling pointer, this may help. Before either problems occurs, memory is created on the heap and the address of this memory block is assigned to a pointer. A leak occurs when the pointer *gets deleted, leaving a block with no reference. In contrast, a dangling pointer occurs when the memory block is deallocated, leaving a pointer that points to deallocated memory. Both are shown in the following figure.*

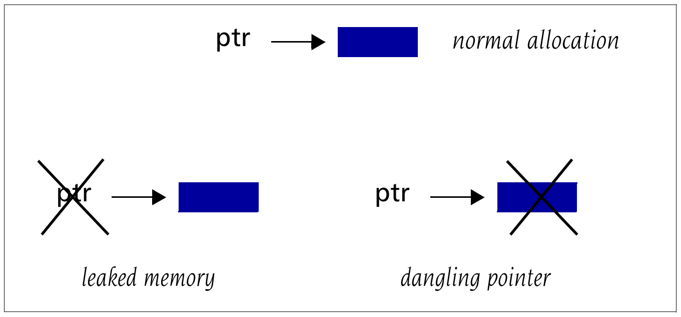


Figure 12: Leaks and Dangling Pointers

The Memory Debugger Leak Detection Page shows all of your program's leaks. For information on detecting leaks, see "*Finding Memory Leaks*" on page 21.

The Heap

The *heap* is an area of memory that your program uses when it wants to dynamically allocate space for data. While using the heap gives you a considerable amount of flexibility, you must manage this resource. You allocate and deallocate this space. In contrast, you do not allocate or deallocate memory in other areas.

Because allocation and deallocation are intimately linked with your program's algorithms and, in some cases, the way you use this memory is implicit rather than explicit, problems associated with the heap are the hardest to find.

Finding Allocation Problems

Memory allocation problems are seldom due to allocation requests. Instead, they occur because your program either is using too much memory or is leaking it. Because an operating system's virtual memory space is large, allocation requests usually succeed. Nevertheless, you should always check the value returned from allocation requests such as **malloc()**, **calloc()**, and **realloc()**. Similarly, you should always check whether the C++ **new** operator returns a null pointer. (Newer C++ compilers throw a **bad_alloc** exception.) If your compiler supports the **new_handler** operator, you can throw your own exception.

You can tell the Memory Debugger to stop execution when your program encounter memory allocation problems. However, since these problems are rare, you might never come across one.

Finding Deallocation Problems

The Memory Debugger can let you know when your program encounters a problem deallocating memory. Some of the problems it can identify are:

- **free not allocated**: An application calls the free() function using an address that is not in a block allocated in the heap.
- **realloc not allocated**: An application calls the realloc() function using an address that is not in a block allocated in the heap.
- **Address not at start of block**: A free() or realloc() function receives a heap address that is not at the start of a previously allocated block.

If a library routine use the memory manager and a problem occurs, the Memory Debugger still locates the problem. For example, the strdup() string library functions call the malloc() function to create memory for a duplicated string. Since the strdup() function is calling the malloc() function, the Memory Debugger can track this memory.

You can tell the Memory Debugger to stop execution just before your program misuses a heap API operation. This lets you see what the problem is before it actually occurs. (For more information, see "*Behind the Scenes*" on page 5.)

Because execution stops before your program's heap manager deallocates memory, you can use the Thread > Set PC *command to set the* PC *to a line after the free request. This means that you can continue debugging past a problem that might cause your program to crash.*

realloc() Problems

The realloc() function can create unanticipated problems. This function can either extend a current memory block, or create a new block and free the old. Although you can check to see which action occurred, you need to code defensively so that problems do not occur. Specifically, you must change every pointer pointing to the memory block to point to the new one. Also, if the pointer doesn't point to the beginning of the block, you need to take some corrective action.

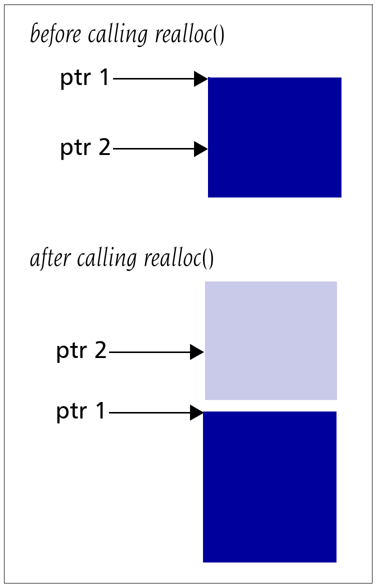
In the following figure, two pointers are pointing to a block. After the realloc() function executes, ptr1 points to the new block. However, ptr2 still points to the original block, a block that was deallocated and returned to the heap manager. (See Figure 13 on page 14.)

Finding Memory Leaks

Technically, there's no such thing as a memory leak. Memory doesn't leak, can't leak. With that said, a memory leak is a block of memory that a program allocates that is no longer referenced. For example, when your program allocates memory, it assigns the block's location to a pointer. A leak can occur if one of the following occurs:

- You assign a different value to that pointer.
- The pointer was a local variable and execution exited from the block.

Figure 13: realloc() Problem



If your program leaks a lot of memory, it can run out of memory. Even if it doesn't run out of memory, your program's memory footprint becomes larger. This increases the amount of paging that occurs as your program executes. Increased paging makes your program run slower.

Here are some of the circumstances in which memory leaks occur:

- **Orphaned ownership**—your program creates memory but does not preserve the address so that it can deallocate it at a later time.

  The following example makes this (extremely) obvious:

  ```
  char *str;
  for( i = 1; i <= 10; i++ )
  {
     str = (char *)malloc(10*i);
  }
  free( str );
  ```

  Within the loop, your program allocates a block of memory and assigns its address to **str**. However, each loop iteration overwrites the address of the previously created block. Because the address of the previously allocated block is lost, its memory can never be made available to your program.

- **Concealed allocation**—the action of creating a memory block is separate from its use.

  As an example, contrast the **strcpy()** and **strdup()** functions. Both do the same thing: they make a copy of a string. However, the **strdup()** function uses the **malloc()** function to create the memory it needs, while the **strcpy()** function uses a buffer that your program creates.

  In general, you must understand what responsibilities you have for allocating and managing memory. For example, when your program receives a handle from a library, the handle allows you to identify a memory block allocated by the library. When you pass the handle back to the library, it knows what memory block contains the data you want to use or manipulate. There may be a considerable amount of memory associated with the handle, and deleting the handle without deallocating the memory associated with the handle leaks memory.

- **Changes in custody**—the routine creating a memory block is not the routine that frees it. (This is related to concealed allocation.)

  For example, routine 2 asks routine 1 to create a memory block. At a later time, routine 2 passes a reference to this memory to routine 3. Which of these blocks is responsible for freeing the block?

  This type of problem is more difficult than other types of problems in that it is not clear when the data is no longer needed. The only thing that seems to work consistently is reference counting. In other words, when routine 2 gets a memory block, it increments a counter. When it passes a pointer to routine 3, routine 3 also increments the counter. When routine 2 stops executing, it decrements the counter. If it is zero, the executing routine frees the memory. If it isn't zero, another routine frees it at another time.
- **Underwritten destructors**:—when a C++ object creates memory, it must ensure that its destructor frees it. No exceptions. This doesn't mean that a block of memory cannot be allocated and used as a general buffer. It just means that when an object is destroyed, it needs to completely clean up after itself.

For more information, see "*Finding free() and realloc() Problems*" on page 17.

Using the Memory Debugger

Here is how you start the TotalView Memory Debugger:

1. Enable the Memory Debugger from within the Memory Debugger Window or the CLI. You must enable the Memory Debugger before execution begins.
2. Tell the Memory Debugger what operations to perform. These operations include hoarding, painting, and telling it to notify you when problems occur using the heap library. *Notification* means that the Memory Debugger stops a program's execution when problems using the heap API occur.

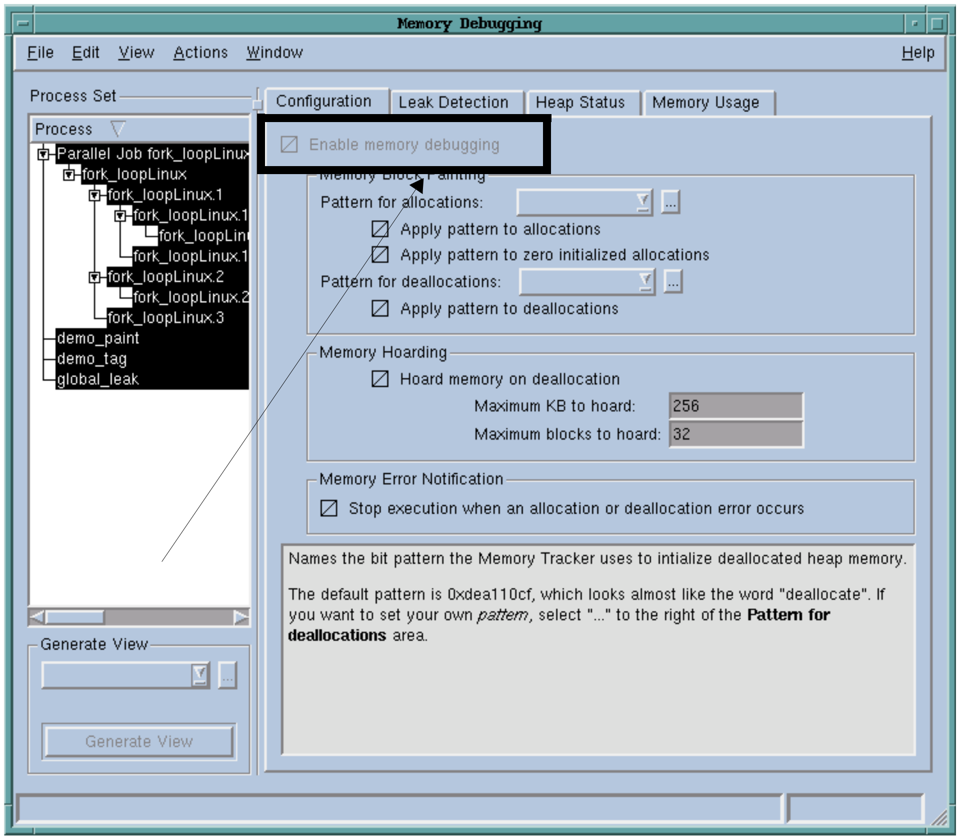
Whenever your program is stopped—for example, it is at a breakpoint or you halted it—you can tell the Memory Debugger to create a view that describes any program leaks or a report that describes currently allocated memory blocks.

Memory Debugger Overview

TotalView must be able to preload your program with the Memory Debugger agent. In many cases, it can do this automatically. However, you must manually link the agent if your application involves remote debugging. In addition, TotalView cannot preload the agent for applications that run on IBM RS/6000 platforms. For more information, see "*Creating Programs for Memory Debugging*" on page 61.

The following procedure describes how you begin using the Memory Debugger:

1 After you start TotalView but before you start executing your program, select the **Tools > Memory Debugging** command. The displayed window shows the Configuration Page.



Names the bit pattern the Memory Tracker uses to intialize deallocated heap memory.

The default pattern is 0xdea110cf, which looks almost like the word "deallocate". If you want to set your own *pattern*, select "..." to the right of the **Pattern for deallocations** area.

Figure 14: Configuration Page

2 Before configuring the Memory Debugger, select one or more of the processes shown in the **Process Set** area on the left.

3 If the **Enable memory debugging** check box isn't checked, you need to select it. If you have explicitly linked your program with the agent, TotalView automatically checks it for you.

4 Start your program and run it to a breakpoint.

Before your program begins execution, you may want to set other options in the Configuration Page:

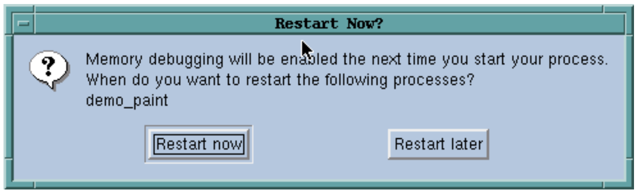
- **Memory Block Painting**—tell the Memory Debugger to paint allocated and deallocated memory and the pattern that the Memory Debugger uses when it paints this memory. For more information, see "*Finding free() and realloc() Problems*" on page 17 and "*Memory Block Painting*" on page 33.
- **Memory Hoarding**—tell the Memory Debugger to hoard deallocated memory blocks, the size of the hoard, and the number of blocks that the hoard can contain. For more information, see "*Memory Hoarding*" on page 35.

- **Memory Error Notification**—tell the Memory Debugger to stop execution and notify you if a heap library problem occurs.

Enabling, Stopping, and Starting

If your program is executing, you cannot enable or disable the Memory Debugger. If you try, TotalView displays its **Restart Now?** Dialog Box:

Figure 15: Restart Now Dialog Box


Restart Now?

Memory debugging will be enabled the next time you start your process.
When do you want to restart the following processes?
demo_paint

Selecting **Restart now** tells TotalView to kill your program, enable the Memory Debugger, and then restart your program. If you select **Restart later**, your program continues executing. After you restart your program, the Memory Debugger will do what you asked it to.

If you turn on notification and all you want to do is stop TotalView from notifying you about heap problems, Remove the check mark from the Configuration Page's **Stop execution when an allocation or deallocation error occurs** check box. While the Memory Debugger continues to track memory events, it no longer stops execution if a problem occurs. Of course, your operating system might terminate execution when an error occurs. However, your program might continue executing. For example, many systems ignore a program to execute a **free()** request that tries to free memory that your program already freed.

Telling the Memory Debugger not to notify you when a problem occurs is useful. For example, suppose you are calling functions in a shared library, and you aren't interested in or can't debug this code and the library has heap problems. Turning off notification lets you execute past this code. Do this by setting a breakpoint at a location after the library function executes. When execution stops, enable notification.

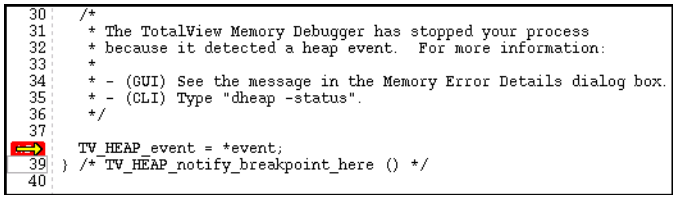
Finding free() and realloc() Problems

The Memory Debugger detects problems that occur when you allocate, reallocate, and free heap memory. This memory is usually allocated by the **malloc()**, **calloc()**, and **realloc()** functions, and deallocated by the **free()** and **realloc()** functions. In C++, the Memory Debugger tracks the **new** and **delete** operators. If your Fortran libraries use the heap API, the Memory Debugger tracks your Fortran program's dynamic memory use. Some Fortran systems use the heap API for assumed-shape, automatic, and allocatable arrays. See your system's **man** pages and other documentation for more information.

Error Notification

After you enable memory debugging and turn on notification, TotalView stops execution if it detects a notifiable event such as a free problem. Execution stops at an internal TotalView breakpoint. As the following figure shows, the lines above the breakpoint have information about what to do next.

Figure 16: TotalView Internal Memory Breakpoint

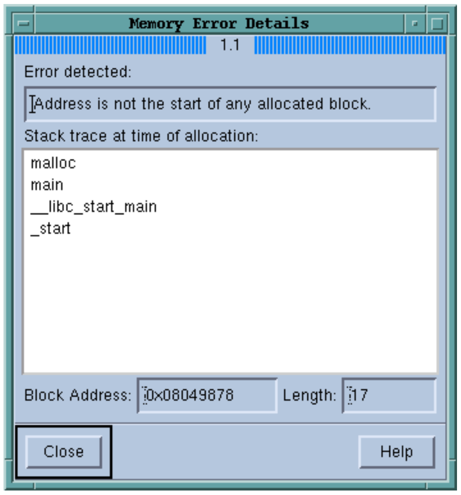


```
 /*
  * The TotalView Memory Debugger has stopped your process
  * because it detected a heap event.  For more information:
  *
  * - (GUI) See the message in the Memory Error Details dialog box.
  * - (CLI) Type "dheap -status".
  */

 TV_HEAP_event = *event;
} /* TV_HEAP_notify_breakpoint_here () */
```

TotalView also displays its **Memory Error Details** Window:

Figure 17: Memory Error Details Window



Error detected:
Address is not the start of any allocated block.
Stack trace at time of allocation:

```
malloc
main
__libc_start_main
_start
```

Block Address: 0x08049878 Length: 17

Notice the following:

- The **Error detection** line tells you what type of error occurred
- The large central area contains a function backtrace if the memory error is related to a block allocated on the heap. These are the stack frames that existed when your program allocated a block. The current backtrace—that is, the backtrace in the Process Window—can be very different.

  If you click on a stack frame in this area, TotalView resets the Process Window to this frame.
- The bottom area contains the block's memory address and its size.

In some cases, the Memory Debugger does not display a backtrace. For example, if you try to free memory allocated on the stack or in a data section, there's no backtrace associated with the memory block. TotalView still displays a Memory Error Details Dialog Box that contains a message such as: "No stack trace available for this memory error."

If you need to redisplay the Memory Error Details Window after you dismiss it, select the **Tools** > **Memory Details** command.

Types of Problems

This section presents some trivial programs that illustrate some of the **free()** and **realloc()** problems that the Memory Debugger detects. The errors shown in these programs are obvious. Errors in your program are, of course, more subtle.

Freeing Unallocated Space

The following section contains programs that free space that they cannot deallocate.

Freeing Stack Memory

The following program allocates stack memory for the **stack_addr** variable. Because the memory was allocated on the stack, the program cannot deallocate it.

```
int main (int argc, char *argv[])
{
   void  *stack_addr  = &stack_addr;
      /* Error: freeing a stack address */
   free(stack_addr);
   return 0;
}
```

Freeing bss Data

The bss section contains uninitialized data. That is, variables in this section have a name and a size but they do not have a value. Specifically, these variables are your program's uninitialized static and global variables. Because they are contained in a data section, your program cannot free their memory.

The following program tries to free a variable in this section:

```
      /* Not initialized; should be in bss */
static int bss_var;

int main (int argc, char *argv[])
{
   void *addr = (void *) (&bss_var);
      /* Error: address in bss section */
   free(addr);
   return 0;
}
```

Freeing Data Section Memory

If your program initializes static and global variables, it places them in your executable's data section. Your program cannot free this memory.

The following program tries to free a variable in this section:

```
       /* Initialized; should be in data section */
static int data_var = 9;

int main (int argc, char *argv[])
{
   void *addr = (void *) (&data_var);
      /* Error: adress in data section */
   free(addr);
   return 0;
}
```

Freeing Memory That Is Already Freed

The following program allocates some memory, then releases it twice. On some operating systems, your program can SEGV on the second free request.

```
int main (int argc, char *argv[])
{
   char *prog_name = argv[0];
   void *s;
      /* Get some memory */
   s = malloc(sizeof(int)*200);
      /* Now release the memory */
   free(s);
      /* Error: Release it again */
   free(s);
   return 0;
}
```

Tracking realloc() Problems

The following program passes a misaligned address to the **realloc()** function.

```
int main (int argc, char *argv[])
{
   char *s, *misaligned_s, *realloc_s;

      /* Get some memory */
   s = malloc(sizeof(int)*64);
      /* Reallocate memory using a misaligned address */
   misaligned_s = s + 8;
   realloc_s = realloc(misaligned_s, sizeof(int)*256));
   return 0;
}
```

In a similar fashion, TotalView detects **realloc()** problems caused by passing addresses to memory sections whose memory cannot be released. For example, TotalView detects problems if you try to do the following:

- Reallocate stack memory.
- Reallocate memory in the data section.
- Reallocate memory in the bss section.

Freeing the Wrong Address

TotalView can detect when a program tries to free a block that does not correspond to the start of a block allocated using the **malloc()** function. The following program illustrates this problem:

```
int main (int argc, char *argv[])
{
   char  *s, *misaligned_s;
```

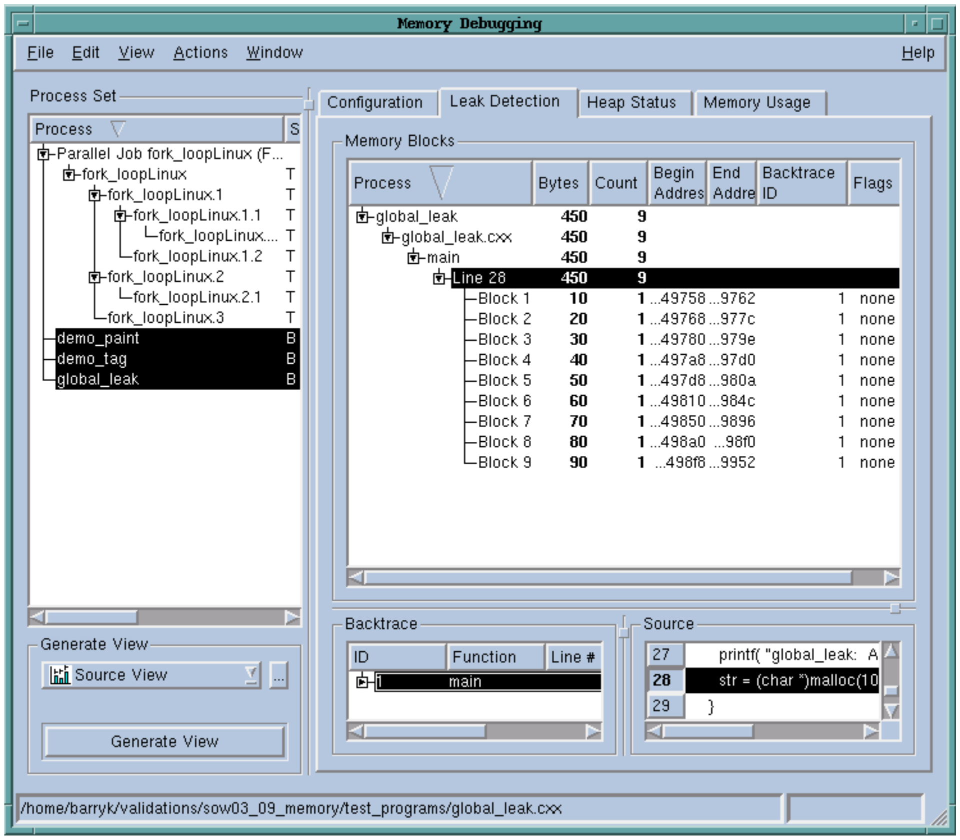
```
        /* Get some memory */
    s = malloc(sizeof(int)*64));
        /*  Release memory using a misaligned address */
    misaligned_s = s + 8;
    free(misaligned_s);
    free(s);
    return 0;
}
```

Finding Memory Leaks

The TotalView Memory Debugger can locate your program's memory leaks and display information about them.

1 Before execution begins, enable the Memory Debugger. (See "*Enabling, Stopping, and Starting*" on page 17.)
2 Run the program and then halt it where you want to look at memory problems. Allow your program to run for a while before stopping execution to give it enough time to create leaks.
3 From the **Memory Debugger** Window (invoked using the **Tools** > **Memory Debugging** command), select the Leak Detection tab. (See Figure 18 on page 21.)

Figure 18: Heap Status Page: Source View



Process	S
Parallel Job fork_loopLinux (F...	
fork_loopLinux	T
fork_loopLinux.1	T
fork_loopLinux.1.1	T
fork_loopLinux....	T
fork_loopLinux.1.2	T
fork_loopLinux.2	T
fork_loopLinux.2.1	T
fork_loopLinux.3	T
demo_paint	B
demo_tag	B
global_leak	B

Process	Bytes	Count	Begin Addres	End Addre	Backtrace ID	Flags
global_leak	**450**	**9**				
global_leak.cxx	**450**	**9**				
main	**450**	**9**				
Line 28	**450**	**9**				
Block 1	**10**	**1**	...49758	...9762	1	none
Block 2	**20**	**1**	...49768	...977c	1	none
Block 3	**30**	**1**	...49780	...979e	1	none
Block 4	**40**	**1**	...497a8	...97d0	1	none
Block 5	**50**	**1**	...497d8	...980a	1	none
Block 6	**60**	**1**	...49810	...984c	1	none
Block 7	**70**	**1**	...49850	...9896	1	none
Block 8	**80**	**1**	...498a0	...98f0	1	none
Block 9	**90**	**1**	...498f8	...9952	1	none

ID	Function	Line #
1	main	

```
27      printf( "global_leak:  A
28      str = (char *)malloc(10
29    }
```

/home/barryk/validations/sow03_09_memory/test_programs/global_leak.cxx

4 Select one or more processes in the **Process Set** area.

5 Select a view within the **Generate View** area and click the **Generate View** button. For example, you might select **Source View**.

6 Examine the list. After you select a leak in the top part of the window, the bottom of the window shows a backtrace of the place where the memory was allocated. After you select a stack frame in the backtrace, TotalView displays the statement where the block was created.

A backtrace is a list of stack frames. The Memory Debugger displays a list that contains the stack frames that existed when you asked the heap manager to allocate memory.

The backtrace that the Memory Debugger displays is the backtrace that existed when your program made the heap allocation request. I*t is not the current backtrace*.

The line number displayed in the Memory Debugger Source Pane is the same line number that TotalView displays in the Process Window Source Pane. If you go to that location, you can begin devising a strategy for fixing the problem. Sometimes you get lucky and the fix is obvious. In most cases, it isn't clear what was (or should be) the last statement to access a memory block. Even if you figure it out, it's extremely difficult to determine if the place you located is really the last place your program needs this data. At this point, it just takes patience to follow your program's logic.

Many users like to generate a view that contains all leaks for the entire program. Do this by setting a breakpoint on your program's **exit** statement. After your program stops executing, generate a Leak Detection View.

You can use the CLI to save the report, as follows:

1 Select the **Tools** > **CLI Window** command.

2 After you see the CLI prompt, capture the **dheap –leaks** output. The following statement shows how you could write this information to a file:

```
exec cat << [ capture dheap -leaks ] > \
          /home/reports/leaks.txt
```

Using Watch Points

For many types of memory problems, identifying where the problem occurred is just the first step. Your next step is to look for the solution. TotalView and the Memory Debugger can help. For example, here's a procedure that lets you identify when your program writes to a memory block:

1 Using the backtrace in the Leak Detection Page, identify where your program allocated the memory.

2 Go to the Process Window and set a breakpoint after that line.

3 Restart your program and run it to that breakpoint.

4 Dive on the pointer and, if it is not automatically dereferenced, dive on the pointer in the Variable Window.

5 Select the **Tools** > **Watchpoint** command and set a watchpoint.

6 Select **Go**.

Your program stops executing when the value contained at this memory location changes. If there are a number of statements in your program that write into this memory location, you might need to select **Go** a number of times. Eventually, you will know when the last time your program changes a value. Watchpoints do not, unfortunately, get triggered when your program reads data.

Fixing Dangling Pointer Problems

Fixing dangling pointer problems is usually more difficult than fixing other memory problems. First of all, you only become aware of them when you realize that the information your program is manipulating isn't what it is supposed to be. Even more troubling, these problems can be intermittent, happening only when your program's heap manager reuses a memory block. For example, if nothing else is running on your computer, the block might never be reused. If there are a large number of jobs running, a deallocated block could be reused quickly.

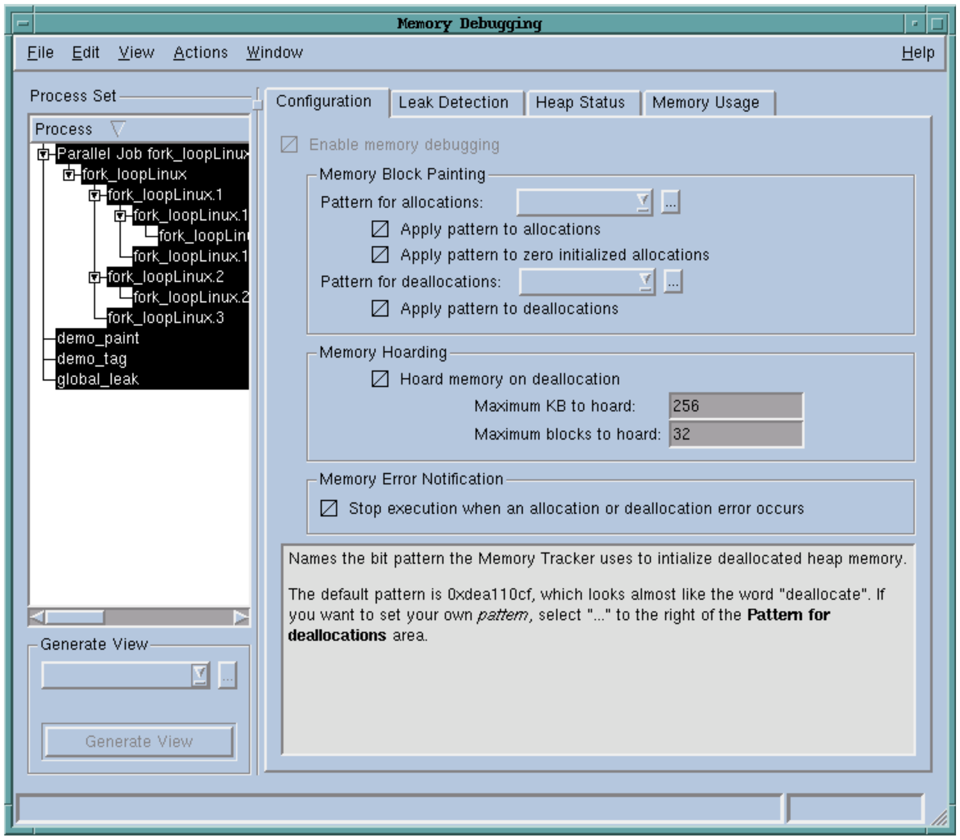
After you identify that you have a dangling pointer problem, you have two problems to solve. The first is to determine where your program freed the memory block. The second is to determine where it *should* free this memory. Memory Debugger tools that can help you are:

- **Block painting**, which tells the Memory Debugger to write a bit pattern into allocated and deallocated memory blocks.
- **Hoarding**, which tells the Memory Debugger to hold onto a memory block when the heap manager receives a request to free it. This is most often used to get beyond where a problem occurs. By allowing the program to continue executing with correct data, you sometimes have a better chance to find the problem. For example, if you also paint the block, it becomes easy to tell what the problem is. In addition, your program might crash. (Crashing while you are in TotalView is a good thing, because TotalView will show the crash point. You immediately know where the problem is.)
- **Watchpoints**, which tell TotalView to stop execution when a new value is written into a memory block. If the Memory Debugger is painting deallocated blocks, you immediately know where your program freed the block.
- The **dheap –tag_alloc** CLI command, which tells TotalView to stop execution when your program deallocates or reallocates memory.

You enable painting and hoarding in the Memory Debugger **Configuration** Page. (See Figure 19 on page 24.)

You can turn painting and hoarding on and off. In addition, you can tell the Memory Debugger what bit patterns to use when it paints memory. For more information, see "*Block Painting*" on page 26.

Figure 19: Configuration Page



Enable memory debugging

Memory Block Painting
Pattern for allocations:
Apply pattern to allocations
Apply pattern to zero initialized allocations
Pattern for deallocations:
Apply pattern to deallocations

Memory Hoarding
Hoard memory on deallocation
Maximum KB to hoard: 256
Maximum blocks to hoard: 32

Memory Error Notification
Stop execution when an allocation or deallocation error occurs

Names the bit pattern the Memory Tracker uses to intialize deallocated heap memory.

The default pattern is 0xdea110cf, which looks almost like the word "deallocate". If you want to set your own *pattern*, select "..." to the right of the **Pattern for deallocations** area.

Dangling Pointers

If you enable memory debugging, TotalView displays information in the Variable Window about the variable's memory use. The following small program allocates a memory block, sets a pointer to the middle of the block, and then deallocates the block:

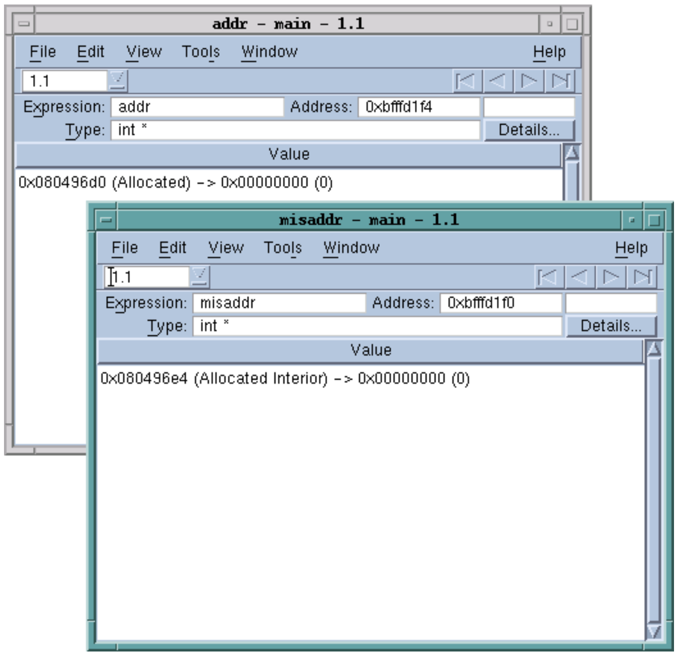
```
main(int argc, char **argv)
{
  int *addr = 0;      /* Pointer to start of block. */
  int *misaddr = 0;   /* Pointer to interior of block. */

  addr = (int *) malloc (10 * sizeof(int));

  misaddr = addr + 5;  /* Point to block interior */

                      /* Deallocate the block. addr and */
                      /* misaddr are now dangling. */
  free (addr);
}
```

The following figure shows two Variable Windows. Execution was stopped before the **free()** function executed. Both windows contain a memory indicator saying that blocks are allocated.


addr – main – 1.1
File Edit View Tools Window Help
1.1
Expression: addr Address: 0xbfffd1f4
Type: int *
Details...
Value

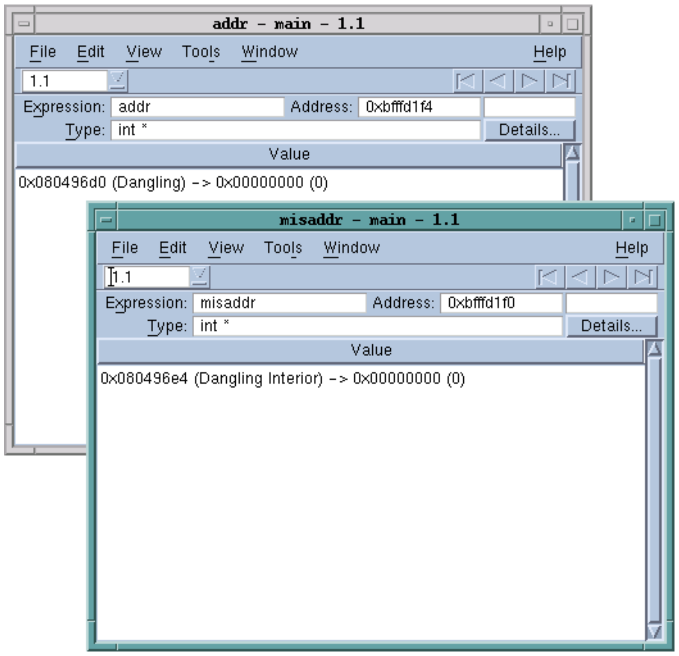
```
0x080496d0 (Allocated) -> 0x00000000 (0)
```

misaddr – main – 1.1
File Edit View Tools Window Help
1.1
Expression: misaddr Address: 0xbfffd1f0
Type: int *
Details...
Value

```
0x080496e4 (Allocated Interior) -> 0x00000000 (0)
```

Figure 20: Allocated Description in a Variable Window

After your program executes the **free()** function, the messages change, as the following figure shows:


addr – main – 1.1
File Edit View Tools Window Help
1.1
Expression: addr Address: 0xbfffd1f4
Type: int *
Details...
Value

```
0x080496d0 (Dangling) -> 0x00000000 (0)
```

misaddr – main – 1.1
File Edit View Tools Window Help
1.1
Expression: misaddr Address: 0xbfffd1f0
Type: int *
Details...
Value

```
0x080496e4 (Dangling Interior) -> 0x00000000 (0)
```

Figure 21: Dangling Description in a Variable Window

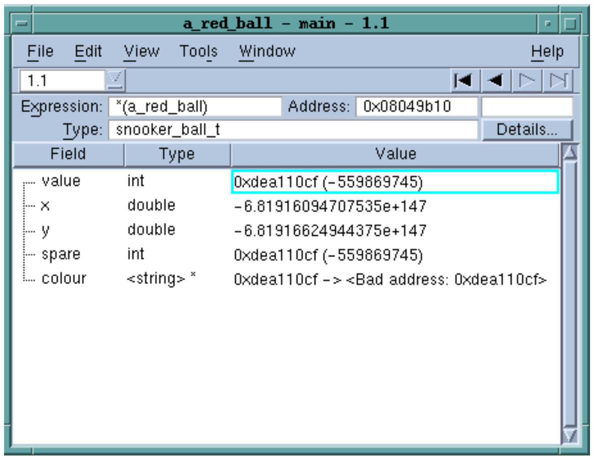
Block Painting

When you enable block painting, TotalView paints a memory block with a bit pattern. You can either specify a pattern or use the default, as follows:

- The default allocation pattern is **0xa110ca7f**, which was chosen because it resembles the word "allocate".
- The default deallocation pattern is **0xdea110cf**, which was chosen because it resembles the word "deallocate". In most cases, you want TotalView to paint memory blocks when they are deallocated.

The following figure shows a variable whose memory was painted:

Figure 22: Block Painting



Expression: *(a_red_ball) Address: 0x08049b10
Type: snooker_ball_t

Field	Type	Value
value	int	0xdea110cf (-559869745)
x	double	-6.81916094707535e+147
y	double	-6.81916624944375e+147
spare	int	0xdea110cf (-559869745)
colour	<string> *	0xdea110cf -> <Bad address: 0xdea110cf>

If the Memory Debugger paints memory for a variable that uses more memory than a word—for example, a double-precision variable—the value that TotalView displays in the Variable Window won't look like the paint pattern. For example, the value in an allocated memory block for a double-precision number is: -6.81916624944375e-147. You need to cast the variable into an array to see this pattern. Or, if you recognize that this number is the pattern, you don't need to cast the value.

Setting the allocation pattern lets you know if your program has initialized a variable. For example, if you display the variable in a Variable Window and see the paint pattern, you'll immediately know that you have a problem.

If you also set a watchpoint on the memory block before your program deallocates it—you might only be able to set it on the first few words of the block—TotalView stops program execution just after the Memory Tracker paints it.

If you are setting a watchpoint on just one element of a structure or an array, you need to dive on the element so that it is the only item in the Variable Window. For example, if you want to set a watchpoint on the **colour** variable in the previous figure, dive on **colour**, and then select the **Tools > Watchpoint** command to set the watchpoint.

If you change the deallocation pattern while your program executes, the pattern lets you know when the block was deallocated. That is, because the Memory Debugger is using a different pattern after you change it, you will know if the memory was allocated or deallocated before or after you made the change.

If you are painting deallocated memory, you could be transforming a working program into one that no longer works. This is good as TotalView will be telling you about a problem.

Hoarding

You can stop your program's memory manager from immediately reusing memory blocks by telling the Memory Debugger to hoard (that is, retain) blocks. Because memory blocks aren't being immediately reused, the data within the blocks isn't being overwritten. This means that your program can continue running with the correct information even though it is accessing deallocated memory. If this weren't the case, any pointers into this memory block would be dangling. In some cases, this uncovers other errors, and these errors can help you track down the problem.

If you are painting and hoarding deallocated memory (and you should be), you might be able to force an error when your program accesses the painted memory.

The Memory Debugger holds onto hoarded blocks for a while before returning them to the heap manager so that the heap manager can reuse them. As the Memory Debugger adds blocks to the hoard, it places them in a first-in, first-out list. When the hoard is full, the Memory Debugger releases the oldest blocks back to your program's memory manager.

You can set the amount of memory that the Memory Debugger hoards using the controls in the Configuration Page.

Using the Memory Debugger Window 2

This chapter examines the Memory Debugger Window. It includes the following topics:

- *"About the Memory Debugger"* on page 29
- *"Configuration Page"* on page 32
- *"Leak Detection Page"* on page 36
- *"Heap Status Page"* on page 40
- *"Memory Usage Page"* on page 42

About the Memory Debugger

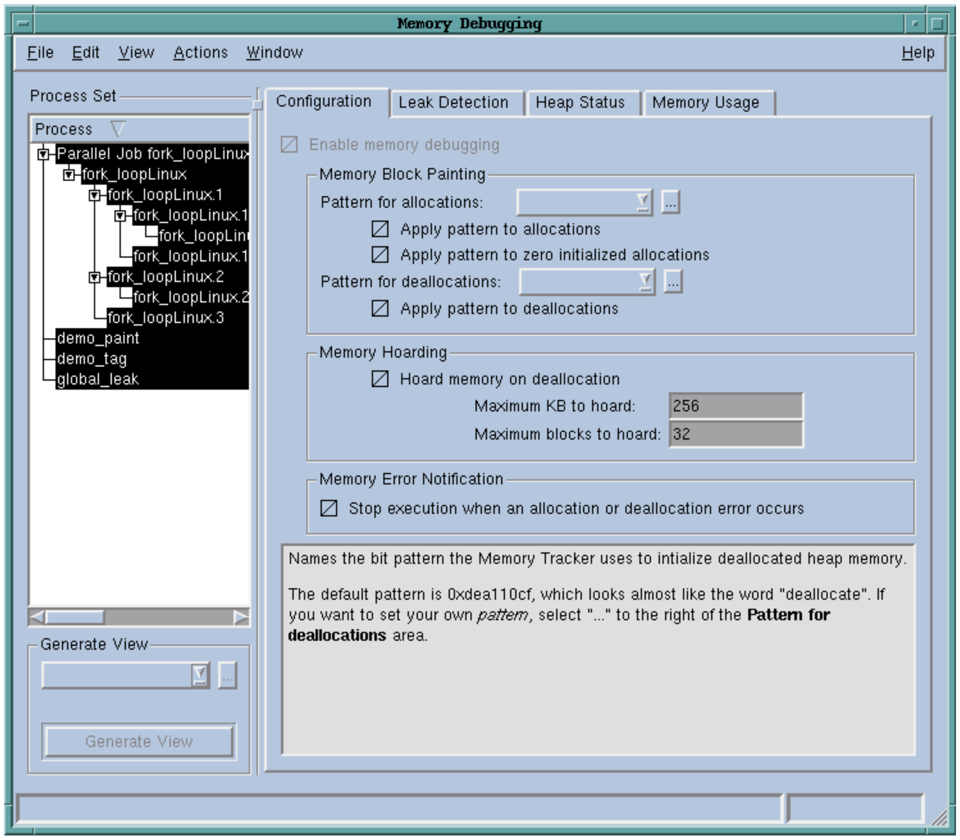
When you configure the Memory Debugger or display a view, the action that the Memory Debugger takes is based on the processes that you select on the left side of the window. (The figure on the next page shows this window.)

The controls in the **Generate View** area tell the Memory Debugger which view to create on the right side of the window. This information is called a *view* because the Memory Debugger just shows a part of the information contained in the Memory Debugger tracking agent. (For information on this agent, see "*Behind the Scenes*" on page 5.)

Process Set Selection

Configuring the Memory Debugger tells it which processes to track and what actions to perform. For example, the Memory Debugger Window shown on the next page can track more than one program. One of these programs has more than one process. If you select three processes out of the twelve processes in this window, a leak detection view only shows leaks from these three processes. It ignores leaks in other processes.

Figure 23: Configuration Page



Enable memory debugging

Memory Block Painting
Pattern for allocations:
Apply pattern to allocations
Apply pattern to zero initialized allocations
Pattern for deallocations:
Apply pattern to deallocations

Memory Hoarding
Hoard memory on deallocation
Maximum KB to hoard: 256
Maximum blocks to hoard: 32

Memory Error Notification
Stop execution when an allocation or deallocation error occurs

Names the bit pattern the Memory Tracker uses to intialize deallocated heap memory.

The default pattern is 0xdea110cf, which looks almost like the word "deallocate". If you want to set your own *pattern*, select "..." to the right of the **Pattern for deallocations** area.

Be careful how many processes you select. With large multiprocess programs, you might be asking the Memory Debugger to process and analyze an enormous amount of data. In most cases, if you select one or two significant processes, you'll receive the information you need. Although the process of generating a view is lengthy, you can redisplay the information quickly after the Memory Debugger creates it.

Generate View

When you are viewing any page except the Configuration Page, you must tell the Memory Debugger which view it should display. The controls in this area of the window are as follows:

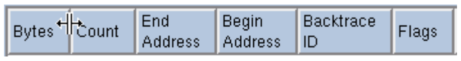
Pulldown list	Select a view from this list. Clicking on the arrow on the right side of this list of views displays your choices.
...	Click this button to display a dialog box that contains preferences that modify or affect a view. The discussion of that page in other sections of this chapter contains information about these preferences.
Generate View	After you select a view, pressing this button tells the Memory Debugger to display the view.

Rows and Columns

If a page displays information in columns, you can resize columns, change the column order, and control which columns the Memory Debugger displays, as follows:

- To resize a column, place the mouse pointer over the vertical column separator in the header. Press your left mouse button and drag the separator so that you've made the column as wide or as narrow as you want it to be. After you finished dragging the separator, release the left mouse button. The following figure shows the second column being made wider:

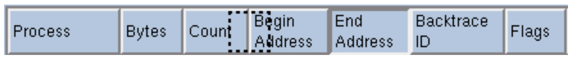
Figure 24: Resizing



  If you double-click on a separator, the Memory Debugger readjusts all widths.

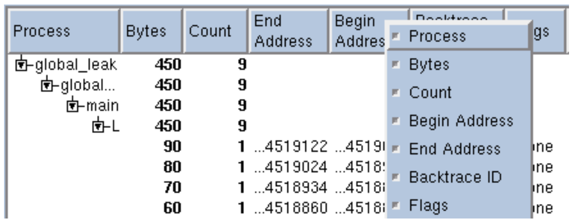
- To change the column order, place your mouse pointer in a column header, press your left mouse button, and then drag the column to its new position. After it is in its new position, release the left mouse button. In the following example, the **Begin Address** column is being moved to the left:

Figure 25: Changing Position



- To tell the Memory Debugger to hide a column or display a column you previously hid, right-click anywhere in the column header area. From the displayed context menu, click on an entry. If the entry is hidden, the Memory Debugger displays it. If the column is displayed, the Memory Debugger hides it. The following figure shows this context menu:

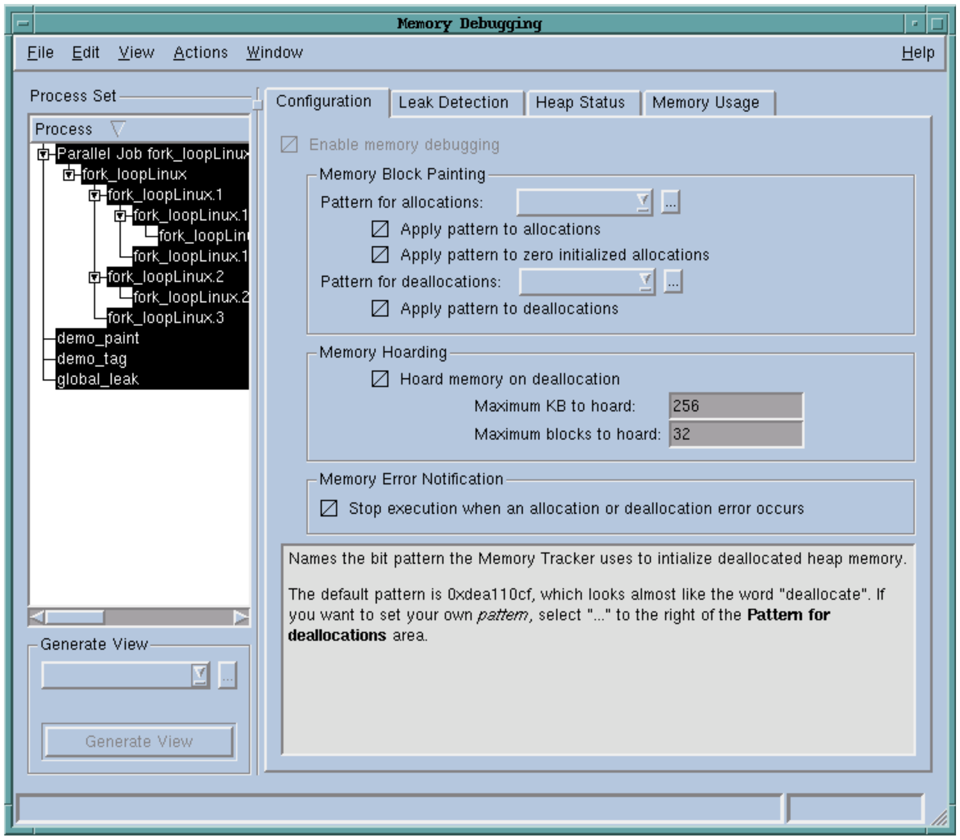
Figure 26: Displaying and Hiding Columns


```
Process          Bytes  Count  End        Begin
                                Address    Addres
global_leak        450      9
  global...        450      9
    main           450      9
      L            450      9
                    90      1  ...4519122 ...4519
                    80      1  ...4519024 ...4518
                    70      1  ...4518934 ...4518
                    60      1  ...4518860 ...4518
```

- To tell the Memory Debugger to sort a column, click on the column heading. You can only sort some columns.

Configuration Page

The controls on the Configuration Page direct the actions that the Memory Debugger performs. The following figure shows this page:



Enable memory debugging

Memory Block Painting

Pattern for allocations:

☐ Apply pattern to allocations

☐ Apply pattern to zero initialized allocations

Pattern for deallocations:

☐ Apply pattern to deallocations

Memory Hoarding

☐ Hoard memory on deallocation

Maximum KB to hoard: 256

Maximum blocks to hoard: 32

Memory Error Notification

☐ Stop execution when an allocation or deallocation error occurs

Names the bit pattern the Memory Tracker uses to intialize deallocated heap memory.

The default pattern is 0xdea110cf, which looks almost like the word "deallocate". If you want to set your own *pattern*, select "..." to the right of the **Pattern for deallocations** area.

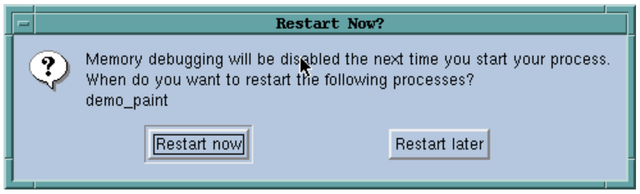
Figure 27: Configuration Page

While you must explicitly tell the Memory Debugger to track your program's use of the heap API, you do not need to enable memory debugging to obtain a Memory Usage View.

The **Enable memory debugging** check box tells the Memory Debugger if it should track your program's use of the heap API. Most computing architectures allow TotalView to enable the Memory Debugger before your program begins executing. However, TotalView cannot directly enable programs that run on an IBM RS/6000 or which run remotely. See Chapter 4, "*Creating Programs for Memory Debugging*," on page 61 for more information. If TotalView can dynamically enable memory debugging, selecting this button loads the Memory Debugger.

You cannot check or uncheck this button while your program is executing. If you try, the Memory Debugger opens a dialog box asking if it should restart your program.

Figure 28: Restart Now Dialog Box



Restart Now?

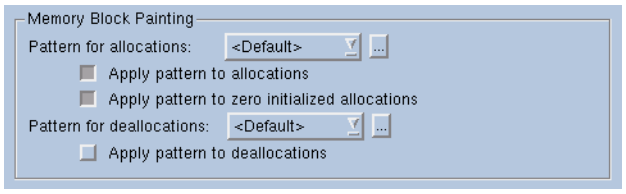
Memory debugging will be disabled the next time you start your process.
When do you want to restart the following processes?
demo_paint

The third line of this error message has the name of the program or process that must be restarted.

Memory Block Painting

When you enable memory block painting, the Memory Debugger writes a bit pattern into newly allocated and newly deallocated heap memory blocks. For information on using block painting, see "*Block Painting*" on page 26.

Figure 29: Memory Block Painting Area



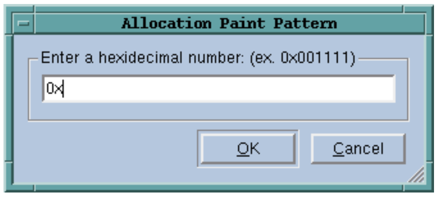
Memory Block Painting
Pattern for allocations: <Default>
Apply pattern to allocations
Apply pattern to zero initialized allocations
Pattern for deallocations: <Default>
Apply pattern to deallocations

Here is a description of these controls:

Pattern for allocations
: The Memory Debugger uses the bit pattern in this box when it paints heap memory that was just deallocated. It uses the same pattern for normal allocations and zero-initialized allocations, which are allocations created by functions such as **calloc()**. The pulldown list contains patterns that you used previously.

When you click the ... button to the right of the pattern box, the Memory Debugger displays a dialog box into which you can type a new pattern:

Figure 30: Allocation Paint Pattern Dialog Box



Allocation Paint Pattern

Enter a hexidecimal number: (ex. 0x001111)
0x

If your program has not started executing, the Memory Debugger might not be able to display a pattern. If it cannot display a pattern, it displays **<Default>**.

You can change this pattern at any time and as many times as you want while your program is executing. Changing the pattern can help you identify when your program allocated a memory block. For example, when you see a pattern, you can tell if it was painted before or after you made a change.

If a data value is greater than the number of bits that the Memory Debugger can paint, TotalView interprets the value using the number of bytes that the variable uses, not the number of bytes in the paint pattern. This means that you might need to cast the displayed value.

If you uncheck this box, the Memory Debugger stops painting allocated memory. You can recheck this box at a later time without having to restart your program.

Apply pattern to allocations

Checking this box tells the Memory Debugger to paint allocated memory using the bit pattern shown in the **Pattern for allocations** text field.

Apply pattern to zero initialized allocations

Checking this box tells the Memory Debugger to paint allocated memory that is set to zero by calls such as **calloc()** using the bit pattern shown in the **Pattern for allocations** text field.

You cannot paint zero-allocated memory unless you are also painting normal allocations. If you remove the check from the **Apply pattern to allocations** box, the Memory Debugger also removes the check from this box.

Enabling this option can break your program if you depend upon the allocated memory being set to zero.

Pattern for deallocations

The Memory Debugger uses the bit pattern in this box when it paints newly deallocated heap memory. For more information, see "*Pattern for allocations" on page* 33.

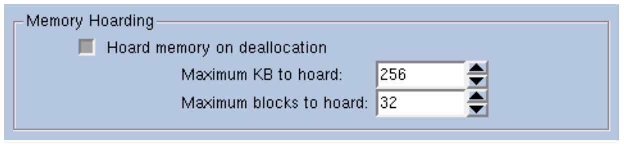
Apply pattern to deallocations

Checking this box tells the Memory Debugger to paint deallocated memory using the bit pattern shown in the **Pattern for deallocations** text field.

Memory Hoarding

The Memory Debugger can delay your program's ability to immediately give back freed memory. This is called *hoarding*. For more information, see "*Hoarding*" on page 27.

Figure 31: Memory Hoarding Area


Memory Hoarding

☐ Hoard memory on deallocation

Maximum KB to hoard: 256

Maximum blocks to hoard: 32

Here is a description of these controls:

Hoard memory on deallocation
: Click on this check box to tell the Memory Debugger to hoard memory. You can check this box while your program is executing.

  If you remove the check while your program is executing, the Memory Debugger no longer hoards newly deallocated blocks. It does not, however, release blocks that it previously retained.

  If the hoard is full and the Memory Debugger needs to hoard a new block, it releases the oldest blocks (that is, those that it first hoarded) so there's enough room in its hoard buffer. You can change the size of the hoard using the next two controls.

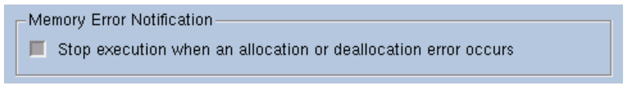
Maximum KB to hoard
: By default, the hoard can grow to 256 KB. You can change the hoard's buffer size by changing this value.

Maximum blocks to hoard
: By default, the hoard can contain up to 32 memory blocks. You can change the number of blocks by changing this value.

Memory Error Notification

If a problem occurs using a function within the heap API, the Memory Debugger can tell TotalView to stop the program's execution so that you can locate the problem. For more information, see "*Finding free() and realloc() Problems*" on page 17.

Figure 32: Memory Error Notification Area

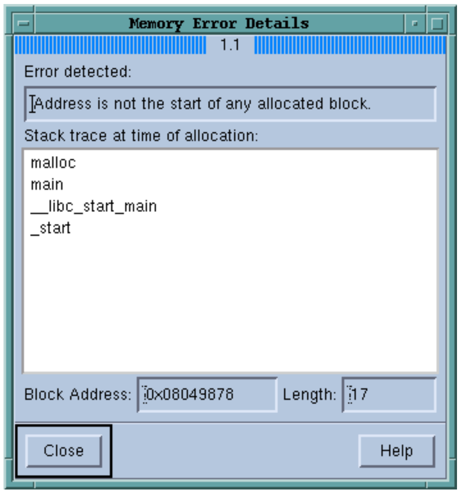

Memory Error Notification

☐ Stop execution when an allocation or deallocation error occurs

Here is a description of this control:

Stop execution when an allocation or deallocation error occurs
: Checking this box tells the Memory Debugger to stop program execution and display a dialog box when it

detects that a problem occurred using the heap API. For example:



Error detected:
Address is not the start of any allocated block.
Stack trace at time of allocation:

```
malloc
main
__libc_start_main
_start
```

Block Address: 0x08049878 Length: 17

Figure 33: Memory Error Details Window

You can turn notification on and off both before and while your program is executing.

Leak Detection Page

The Memory Debugger can display information about the leaks it discovers in two ways: using a Source View or a Backtrace View. Each view displays approximately the same information.

Be careful how many processes you select. With large multiprocess programs, you might be asking the Memory Debugger to process and analyze an enormous amount of data. In most cases, if you select one or two significant processes, you'll receive the information you need. Although the process of generating a view is lengthy, you can redisplay the information quickly after the Memory Debugger creates it.

Source View

The Source View organizes the leaks in your program by the program, routine, file, and block.

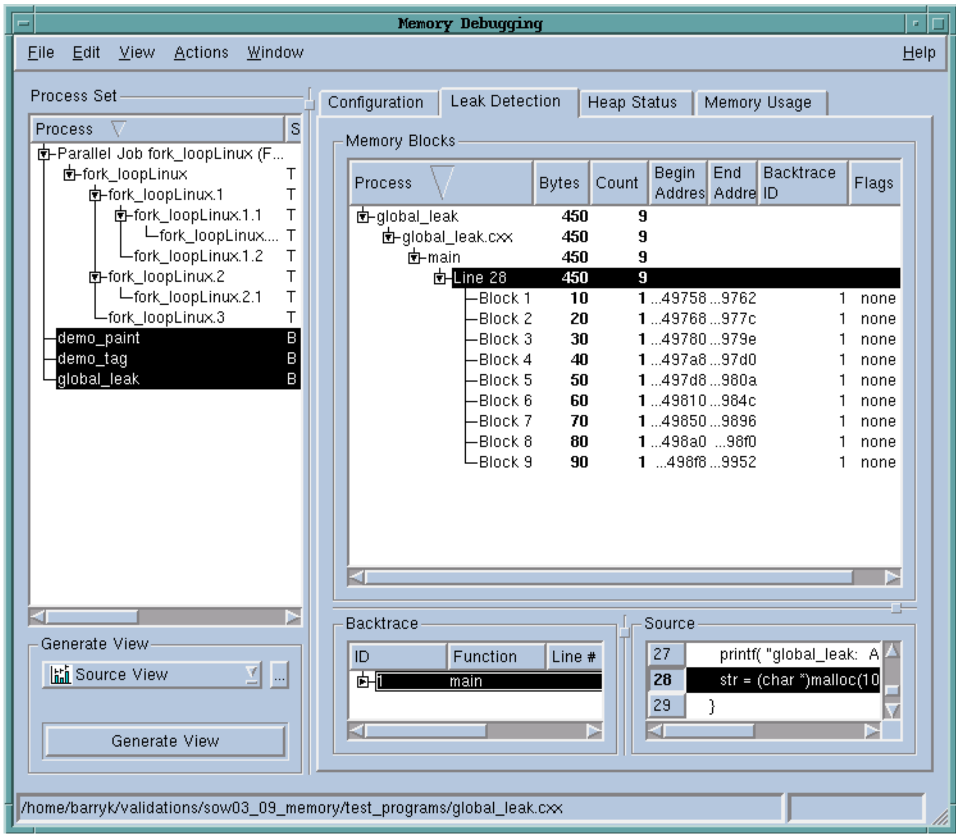
To create this view:

- Select the processes for which you want information in the **Process Set** area.
- Select **Source View**, and then select **Generate View**.

In this view, the first column, **Process**, contains a hierarchical display organizing your program's information. The Backtrace and Source Panes contain additional information about the line you select in the Memory Blocks Pane. In other words, this view organizes the information in the same way that your program is organized.

The following figure shows a Source View.

Figure 34: Leak Detection Page: Source View



Process	S
Parallel Job fork_loopLinux (F...	
fork_loopLinux	T
fork_loopLinux.1	T
fork_loopLinux.1.1	T
fork_loopLinux....	T
fork_loopLinux.1.2	T
fork_loopLinux.2	T
fork_loopLinux.2.1	T
fork_loopLinux.3	T
demo_paint	B
demo_tag	B
global_leak	B

Process	Bytes	Count	Begin Addres	End Addre	Backtrace ID	Flags
global_leak	450	9				
global_leak.cxx	450	9				
main	450	9				
Line 28	450	9				
Block 1	10	1	...49758	...9762	1	none
Block 2	20	1	...49768	...977c	1	none
Block 3	30	1	...49780	...979e	1	none
Block 4	40	1	...497a8	...97d0	1	none
Block 5	50	1	...497d8	...980a	1	none
Block 6	60	1	...49810	...984c	1	none
Block 7	70	1	...49850	...9896	1	none
Block 8	80	1	...498a0	...98f0	1	none
Block 9	90	1	...498f8	...9952	1	none

ID	Function	Line #
1	main	

```
27    printf( "global_leak:  A
28    str = (char *)malloc(10
29    }
```

/home/barryk/validations/sow03_09_memory/test_programs/global_leak.cxx

The bottom-most rows in the hierarchy contain information about an individual leak. As you go up the tree towards the process name, the Memory Debugger summarizes the number of bytes and the number of leaks associated with the information at lower levels of the tree. In this example, the program leaked 450 bytes and 9 allocations were associated with leaks. Because each block has the same backtrace ID (1), you can tell that they all came from the same location in the program. If the numbers were different, the same routine might still be implicated. However, a different backtrace number indicates that the routine was invoked in a different way.

When you click on a line in the Memory Blocks Pane, the Memory Debugger shows information in the Backtrace Pane, as follows:

- The backtrace being displayed is the one that existed when your program allocated the memory block. The Memory Debugger highlights the frame that it thinks is the one you should be focusing on. That is, it highlights where the memory allocation was made. If it guesses wrong, you can reset the hierarchy of backtraces by right-clicking your mouse on the backtrace that you want displayed, as follows.

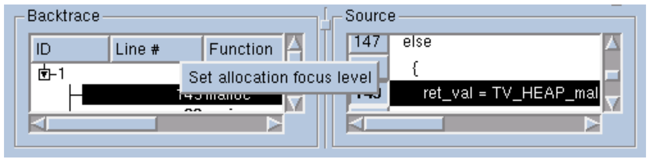


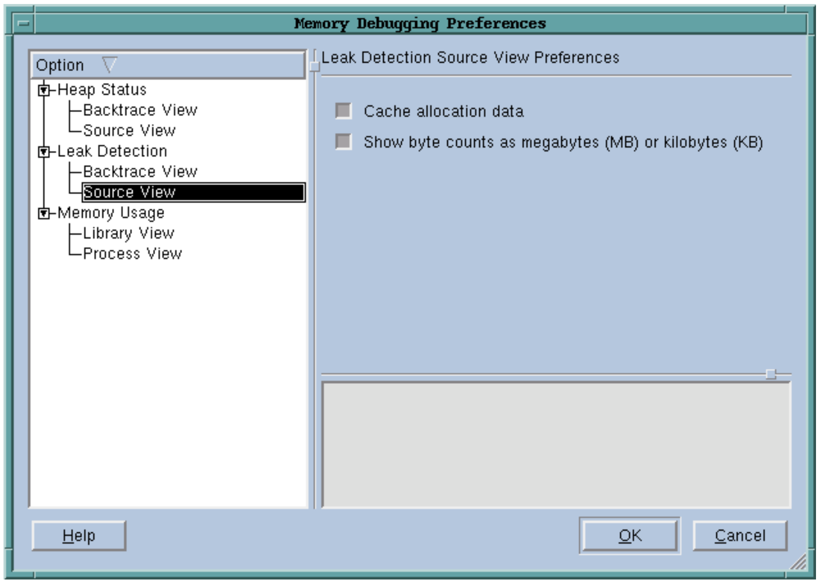
Figure 35: Backtrace and Source Panes

From the context menu, select **Set allocation focus level**.

For example, assume that you have created a function named **my_malloc()** that filters all of your memory allocations. The Memory Debugger would probably guess that this is the function to highlight in the Backtrace Pane. However, you probably want to set the allocation focus on the function that called **my_malloc()**. Do this by selecting that function, and then right-clicking on it to invoke the **Set allocation focus level** command.

- The Source Pane shows the line in your program that contained the memory allocation statement. When you click on a backtrace ID, the Memory Debugger updates the Source Pane to show the line. The line number associated with this line is the same line number that appears in the Process Window Source Pane.

You can set two preferences for Leak Detection views. After displaying the preferences dialog box, the Memory Debugger displays the following dialog box:



☐ Cache allocation data
☐ Show byte counts as megabytes (MB) or kilobytes (KB)

Figure 36: Leak Detection Source View Preferences

To set preferences associated with the Source View, select the ... button within the Generate View area on the left. The preferences are as follows:

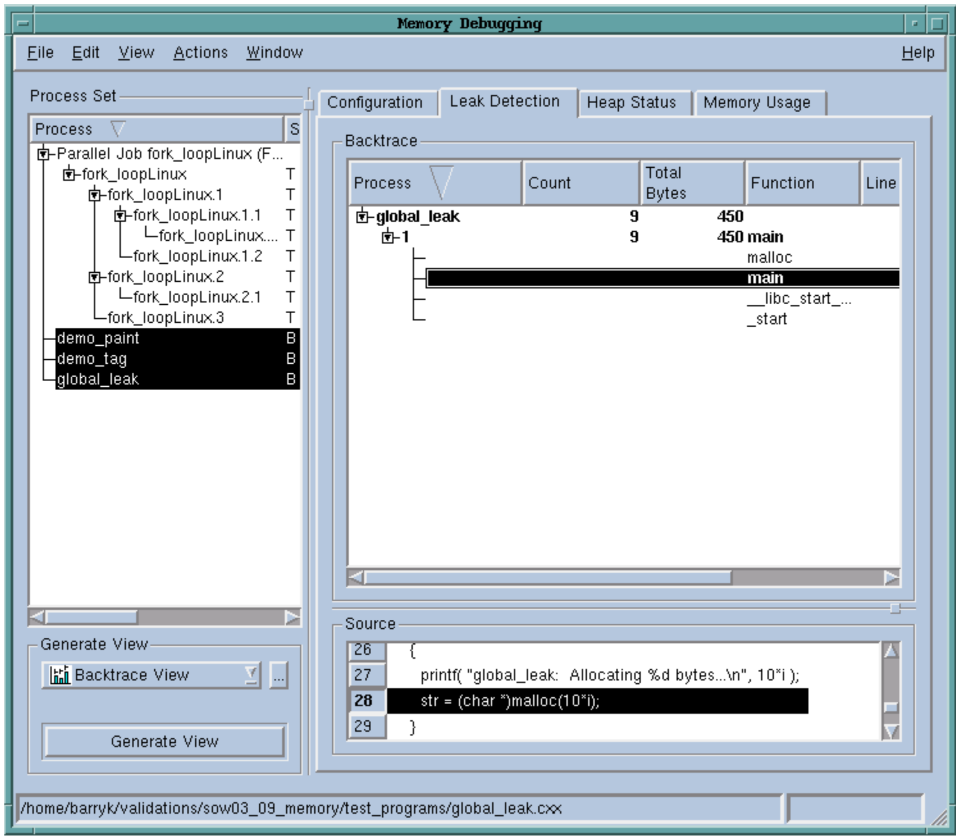
Cache allocation data
: For small-to-medium sized programs, you can increase performance by caching allocation information. If the program is large or has many processes, turn off caching, as it consumes a large amount of memory. However, turning off caching means that the Memory Debugger fetches allocation data each time you generate a view.

Show byte counts as megabytes (MB) or kilobytes (KB)
: By default, the Memory Debugger displays memory sizes in KB. Selecting this check box tells the Memory Debugger to choose the most convenient size.

Backtrace View

The Backtrace View organizes the leaks in your program by the backtrace number created by the Memory Debugger. To create this view, select **Backtrace View**, and then select **Generate View**. In this view, the first column, **Process**, has a numeric list of all the backtrace ID numbers that the Memory Debugger creates.

Figure 37: Leak Detection Page: Backtrace View


Memory Debugging
File Edit View Actions Window Help
Process Set
Configuration Leak Detection Heap Status Memory Usage
Backtrace
Source
Generate View
Backtrace View
Generate View

Process	S
Parallel Job fork_loopLinux (F...	
fork_loopLinux	T
fork_loopLinux.1	T
fork_loopLinux.1.1	T
fork_loopLinux....	T
fork_loopLinux.1.2	T
fork_loopLinux.2	T
fork_loopLinux.2.1	T
fork_loopLinux.3	T
demo_paint	B
demo_tag	B
global_leak	B

Process	Count	Total Bytes	Function	Line
global_leak	9	450		
1	9	450	main	
			malloc	
			main	
			__libc_start_...	
			_start	

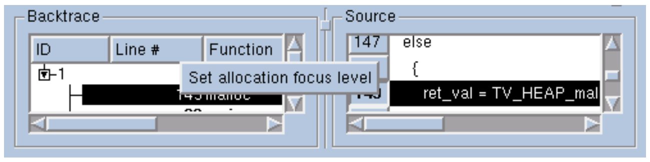
```
{
  printf( "global_leak:  Allocating %d bytes...\n", 10*i );
  str = (char *)malloc(10*i);
}
```

/home/barryk/validations/sow03_09_memory/test_programs/global_leak.cxx

When you look at one backtrace, you might be seeing the rolling together of many leaks into one. You can tell how many leaks are associated with one ID by looking at the **Count** column. In this example, nine leaks are associated with backtrace ID 1.

When you click on a line having a source code associated with it, the Memory Debugger displays that line in its Source Pane.

The backtrace being displayed is the one that existed when your program allocated the memory block. The Memory Debugger highlights the frame that it thinks is the one you should be focusing on. That is, it highlights where the memory allocation was made. If it guesses wrong, you can reset the hierarchy of backtraces by right-clicking your mouse on the back trace that you want displayed, as follows.



```
else
  {
  ret_val = TV_HEAP_mal
```

Figure 38: Backtrace and Source Panes

From the context menu, select **Set allocation focus level**.

For example, assume that you have created a function named **my_malloc()** that filters all of your memory allocations. The Memory Debugger would probably guess that this is the function to highlight in the Backtrace Pane. However, you probably want to set the allocation focus on the function that called **my_malloc()**. Do this by selecting that function, and then right-clicking on it to invoke this command.

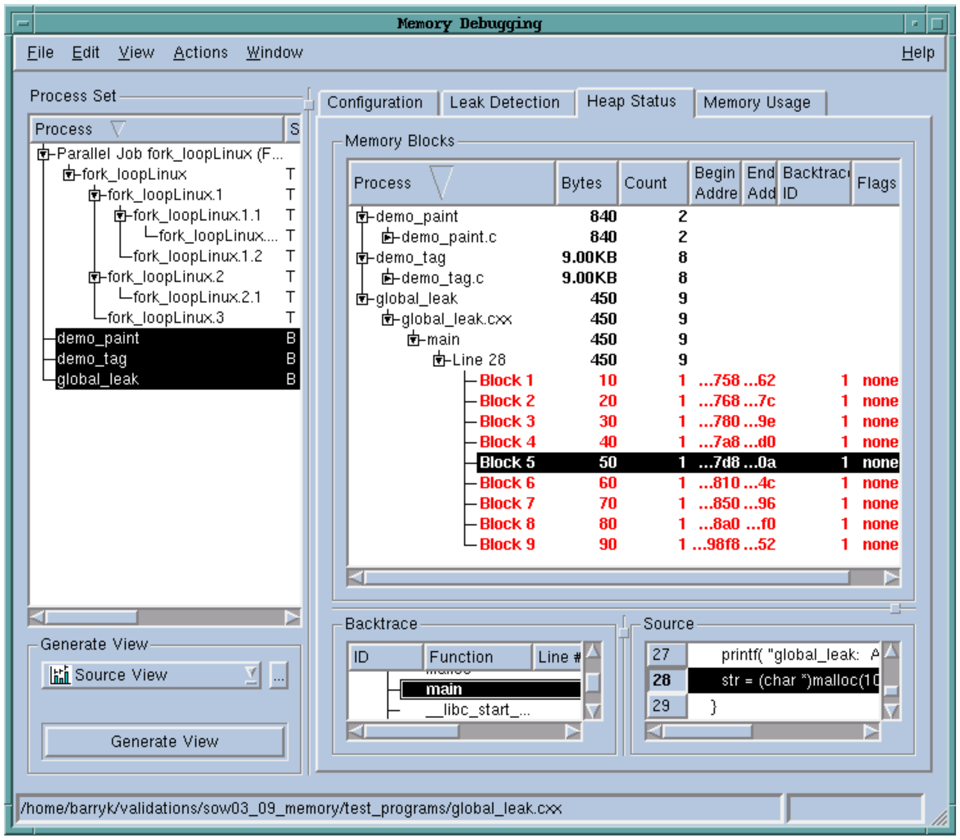
To set preferences associated with the Backtrace View, select the … button within the Generate View area on the left. The preferences are as follows:

Cache allocation data
For small-to-medium sized programs, you can increase performance by caching allocation information. If the program is large or has many processes, turn off caching, as it consumes a large amount of memory. However, turning off caching means that the Memory Debugger fetches allocation data each time you generate a view.

Show byte counts as megabytes (MB) or kilobytes (KB)
By default, the Memory Debugger displays memory sizes in KB. Selecting this check box tells the Memory Debugger to choose the most convenient size.

Heap Status Page

The Heap Status Page displays information about all memory blocks that your program has not yet freed. The views shown in this page can be quite large. Like the Leak Detector Page, you can tell the Memory Debugger to display a Source View or a Backtrace View. The following figure shows a Heap Status Source View



Process	S
Parallel Job fork_loopLinux (F...	
fork_loopLinux	T
fork_loopLinux.1	T
fork_loopLinux.1.1	T
fork_loopLinux....	T
fork_loopLinux.1.2	T
fork_loopLinux.2	T
fork_loopLinux.2.1	T
fork_loopLinux.3	T
demo_paint	B
demo_tag	B
global_leak	B

Memory Blocks

Process	Bytes	Count	Begin Addre	End Add	Backtrac ID	Flags
demo_paint	840	2				
demo_paint.c	840	2				
demo_tag	9.00KB	8				
demo_tag.c	9.00KB	8				
global_leak	450	9				
global_leak.cxx	450	9				
main	450	9				
Line 28	450	9				
Block 1	10	1	...758	...62	1	none
Block 2	20	1	...768	...7c	1	none
Block 3	30	1	...780	...9e	1	none
Block 4	40	1	...7a8	...d0	1	none
Block 5	50	1	...7d8	...0a	1	none
Block 6	60	1	...810	...4c	1	none
Block 7	70	1	...850	...96	1	none
Block 8	80	1	...8a0	...f0	1	none
Block 9	90	1	...98f8	...52	1	none

ID	Function	Line #
	main	
	__libc_start_...	

```
27      printf( "global_leak: A
28      str = (char *)malloc(1(
29      }
```

/home/barryk/validations/sow03_09_memory/test_programs/global_leak.cxx

Figure 39: Heap Status Page: Source View

In most cases, an individual item is not very remarkable or noteworthy. However, the "rolled-up" information about your allocations can help you better understand your program's behavior.

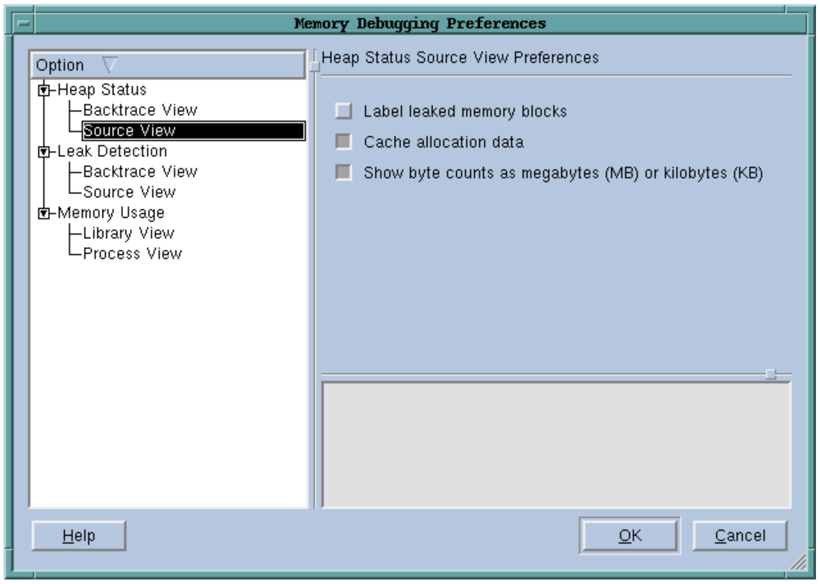
For example, if your program's size is greater than you'd expect it to be, you can select the **Bytes** column so that the largest allocations are all grouped together. Concentrating on the statements allocating this memory should lead you to the problem's solution.

Similarly, if your program is allocating lots of little blocks of memory, these allocations might be hurting performance. Looking at the information in the **Bytes** and **Count** columns might also give you some hints about where you can improve performance.

The Heap Status and Leak Detection Views contain the same type of information. The only differences between these views and what you see in Heap Status View are:

- These views contain all memory allocations, not just allocations that represent leaks.
- If you select the **Label leaked memory blocks** preference, the Memory Debugger analyzes your heap allocations to see if the allocation represents

a leak. Although it displays all allocations, it displays leaked allocations in color.



Heap Status Source View Preferences

- Label leaked memory blocks
- Cache allocation data
- Show byte counts as megabytes (MB) or kilobytes (KB)

Figure 40: Heap Status Preferences: Source View

Be careful how many processes you select. With large multiprocess programs, you might be asking the Memory Debugger to process and analyze an enormous amount of data. In most cases, if you select one or two significant processes, you'll receive the information you need. Although the process of generating a view is lengthy, you can redisplay the information quickly after the Memory Debugger creates it.

Memory Usage Page

The Memory Usage Page tells you how your program is using memory, and where this memory is being used. One way to use this page is to compare memory use over time, so that you can tell if your program is leaking memory. If a program is leaking memory, you'll see that the amount of memory being used steadily increases over time. You can also compare memory use between processes, which can tell you if a process is using more memory than you expect.

You do not need to enable memory debugging to obtain a Memory Usage View.

The Memory Debugger can present either a Process or Library View. The following figure shows an example of a Process View.

Clicking on a column header sorts the information from maximum to minimum, or vice versa.

If you add the memory values of all columns except the last, the sum doesn't equal the last column's value. There are several reasons for this. For example, most operating systems divide segments into pages, and information in a segment does not cross page boundaries. Another reason

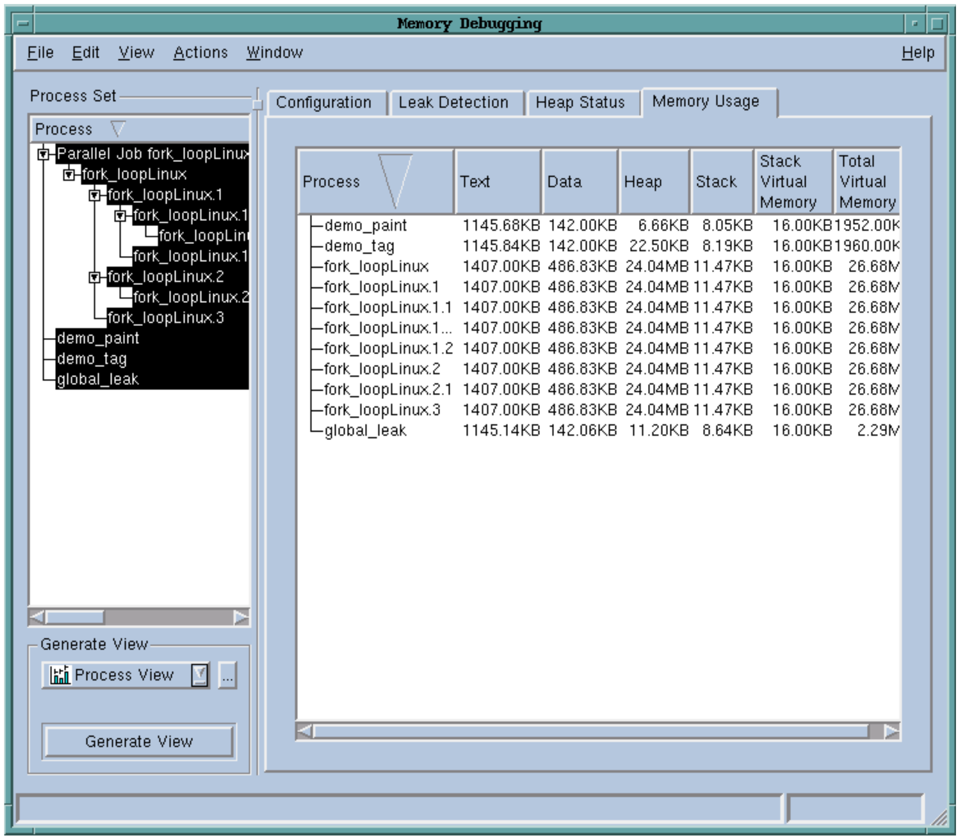


Figure 41: Memory Usage Page: Process View

Process	Text	Data	Heap	Stack	Stack Virtual Memory	Total Virtual Memory
demo_paint	1145.68KB	142.00KB	6.66KB	8.05KB	16.00KB	1952.00K
demo_tag	1145.84KB	142.00KB	22.50KB	8.19KB	16.00KB	1960.00K
fork_loopLinux	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1...	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.1.2	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.2	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.2.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
fork_loopLinux.3	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68M
global_leak	1145.14KB	142.06KB	11.20KB	8.64KB	16.00KB	2.29M

is that a process could map a file or an anonymous region. Areas such as these are part of what appear in the Stack Virtual Memory column. However, they do not appear elsewhere.

The information in these columns is as follows:

Process	The name of your process.
Text	The amount of memory used to store your program's machine code instructions.
Data	The amount of memory used to store uninitialized and initialized data.
Heap	The amount of memory currently being used for data created at run time.
Stack	The amount of memory used by the currently executing routine and all the routines in its backtrace.

If you are looking at a multi-threaded process, TotalView only shows information for the main thread's stack. The stack size of some threads does not change over time on some architectures.

On some systems, the space allocated for a thread is considered part of the heap.

Stack Virtual Memory

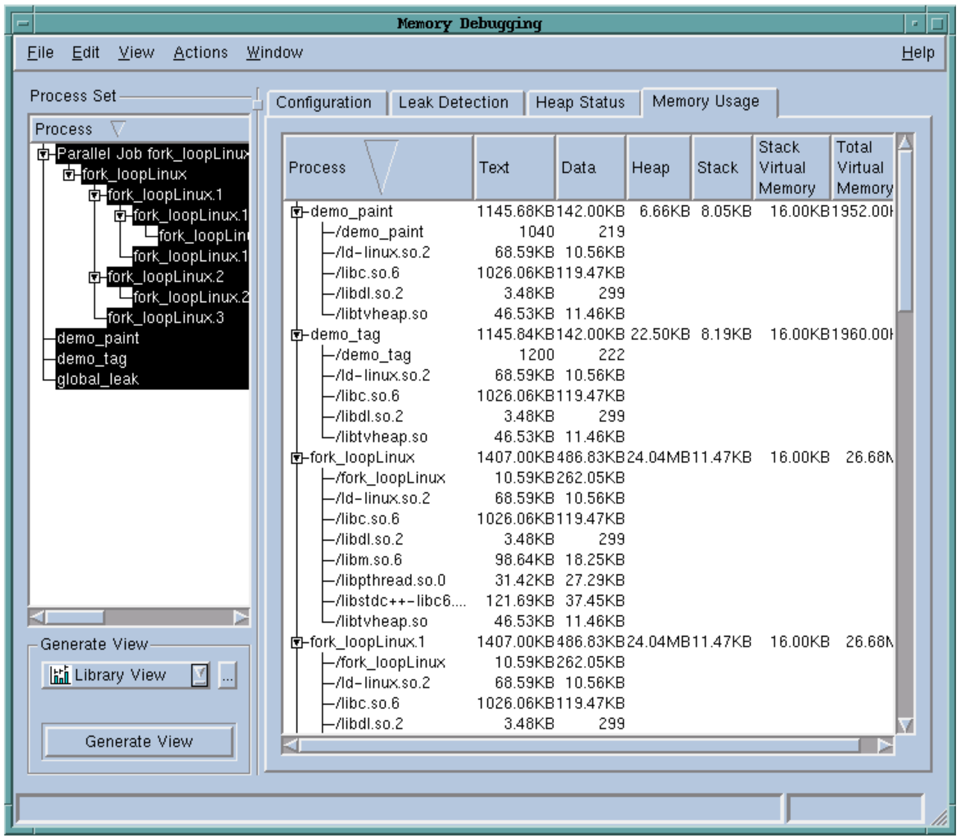
The logical size of the stack. This value is the difference between the current value of the stack pointer and the value reported in the **Stack** column. This value can differ from the size of the virtual memory mapping in which the stack resides.

Total Virtual Memory

The sum of the sizes of the mappings in the process's address space.

The Library Pane shows which library files are contained within your executable. In addition to the same kind of information shown in the Process View, this view shows the amount of memory used by the text and data segments of these libraries. (See the following figure.)

Figure 42: Memory Usage View: Library View



Process	Text	Data	Heap	Stack	Stack Virtual Memory	Total Virtual Memory
demo_paint	1145.68KB	142.00KB	6.66KB	8.05KB	16.00KB	1952.00
/demo_paint	1040	219				
/ld-linux.so.2	68.59KB	10.56KB				
/libc.so.6	1026.06KB	119.47KB				
/libdl.so.2	3.48KB	299				
/libtvheap.so	46.53KB	11.46KB				
demo_tag	1145.84KB	142.00KB	22.50KB	8.19KB	16.00KB	1960.00
/demo_tag	1200	222				
/ld-linux.so.2	68.59KB	10.56KB				
/libc.so.6	1026.06KB	119.47KB				
/libdl.so.2	3.48KB	299				
/libtvheap.so	46.53KB	11.46KB				
fork_loopLinux	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68
/fork_loopLinux	10.59KB	262.05KB				
/ld-linux.so.2	68.59KB	10.56KB				
/libc.so.6	1026.06KB	119.47KB				
/libdl.so.2	3.48KB	299				
/libm.so.6	98.64KB	18.25KB				
/libpthread.so.0	31.42KB	27.29KB				
/libstdc++-libc6....	121.69KB	37.45KB				
/libtvheap.so	46.53KB	11.46KB				
fork_loopLinux.1	1407.00KB	486.83KB	24.04MB	11.47KB	16.00KB	26.68
/fork_loopLinux	10.59KB	262.05KB				
/ld-linux.so.2	68.59KB	10.56KB				
/libc.so.6	1026.06KB	119.47KB				
/libdl.so.2	3.48KB	299				

Using the dheap Command

3

The **dheap** command lets you track memory problems from within the CLI. Although the **dheap** command lets you do everything that you can do using the GUI, there are also a few things that are unique to the CLI. The following list presents actions that you can perform in both:

- To see the status of the Memory Debugger, use the **dheap** command.
- To display information about the heap, use the **dheap –info** command. You can show information for the entire heap or limit what TotalView displays to just a part of it.
- To enable and disable the Memory Debugger, use the **dheap –enable** and **dheap –disable** commands.
- To start and stop error notification, use the **dheap –notify** and **dheap –nonotify** commands.
- To check for leaks, use the **dheap –leaks** command.
- To paint memory with a bit pattern, use the **dheap –paint** command.
- To hoard memory, use the **dheap –hoard** command.

There are several **dheap** *options not yet available in the* GUI.

dheap Example

The following example shows the kind of information that the CLI displays after the Memory Debugger locates an error:

```
d1.<> dheap
             process:   Enable   Notify    Available
    1         (18993):     yes      yes          yes
    1.1    realloc: Address does not match any allocated
block.: 0xbfffd87c
```

```
d1.<> dheap -info -backtrace
process     1   (18993):
       0x8049e88 --     0x8049e98       0x10 [         16]
  flags: 0x0 (none)
    :   realloc  PC=0x400217e5 [/.../malloc_wrappers_dlopen.c]
    :   argz_append   PC=0x401ae025 [/lib/i686/libc.so.6]
    :   __newlocale   PC=0x4014b3c7 [/lib/i686/libc.so.6]
    :
...
.../malloc_wrappers_dlopen.c]
    :   main     PC=0x080487c4 [../realloc_prob.c]
    :   __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
    :   _start   PC=0x08048621 [/.../realloc_prob]

       0x8049f18 --     0x8049f3a       0x22 [         34]
  flags: 0x0 (none)
    :   realloc  PC=0x400217e5 [/.../malloc_wrappers_dlopen.c]
    :   main     PC=0x0804883e [../realloc_prob.c]
    :   __libc_start_main PC=0x40140647 [/lib/i686/libc.so.6]
    :   _start   PC=0x08048621 [/.../realloc_prob]
```

The information that is displayed in this example is explained in more detail later in this chapter.

dheap

Controls heap debugging

Format:

Shows Memory Debugger state

dheap [–status]

Shows information about a backtrace

dheap –backtrace [*subcommands* **]**

Enables or disables the Memory Debugger

dheap { –enable | –disable }

Enables or disables the retaining (hoarding) of freed memory blocks

dheap –hoard [*subcommands* **]**

Displays Memory Debugger information

dheap –info [*subcommands* **]**

Indicates whether an address is in a deallocated block

dheap –is_dangling *address*

Locates memory leaks

dheap –leaks [–check_interior]

Enables or disables Memory Debugger event notification

dheap –[no]notify

Paints memory with a distinct pattern

dheap –paint [*subcommands* **]**

Enables or disables allocation and reallocation notification

dheap –tag_alloc *subcommand* **[** *start_address* **[** *end_address* **]]**

Displays the Memory Debugger version number

dheap –version

Arguments:

[–status]
Displays the current state of the Memory Debugger. This tells you if a process is capable of having its heap operations traced and if TotalView will notify you if a notifiable heap event occurs. If TotalView stops a thread because one of these events occur, it displays information about this event.

If you do not use other options to the **dheap** command, you can omit this option.

–backtrace [*subcommands* **]**
Shows the current settings for the backtraces associated with a memory allocation. This information includes the *depth* and the *trim* (described later in this section).

–status
Tells TotalView to display backtrace information. If you do not use other backtrace options, you can omit this option.

–set_depth *depth*
–reset_depth
Set or reset the *depth*. The *depth* is the maximum number of PCs that the Memory Debugger includes when it creates a backtrace. (The backtrace is created when a memory block is allocated or reallocated.) The *depth* value starts after the *trim* value. That is, the number of excluded frames does not include the trimmed frames.

When you use the –**reset_depth** option, TotalView either restores its default setting or the setting you set using the **TV_HEAP_ARGS** environment variable.

–set_trim *trim*
–reset_trim
Sets or resets the *trim*. The *trim* describes the number of PCs from the top of the stack that the Memory Debugger ignores when it creates a backtrace. As the backtrace includes procedure calls from within the Memory Debugger, setting a trim value removes them from the backtrace. The default is to exclude Memory Debugger procedures. Similarly, your program might call the heap manager from within library code. If you do not want to see call frames showing a library, you can exclude them.

When you use the –**reset_trim** option, TotalView either restores its default setting or the setting you set using the **TV_HEAP_ARGS** environment variable.

–display *backtrace_id*
Displays the stack frames associated with the backtrace identified by *backtrace_id*.

–enable/–disable
Using the –**enable** option tells TotalView to use the Memory Debugger agent to record heap events the next time you start the program. Using the –**disable** option tells TotalView to not use the agent the next time you start your program.

If necessary, you must preload the agent (see Chapter 4, "*Creating Programs for Memory Debugging*," on page 61 for information) before using this option.

–hoard [*subcommands*]
Tells the Memory Debugger not to surrender allocated blocks back to your program's heap manager. If you do not type a subcommand, the Memory Debugger displays information about the hoarded blocks. For more information, see "*Memory Reuse: dheap –hoard*" on page 55.

[–status]
Displays hoard settings. Information displayed indicates if hoarding is enabled, if deallocated blocks are added to the hoard (or only those that are tagged), the maximum size of the hoard, and the hoard's current size.

If you do not use other hoarding options, you can omit the –**status** option when you want to see status information.

–**display** [*start_address* [*end_address*]]
Displays the contents of the hoard. You can restrict the display by specifying *start_address* and *end_address*. If you omit *end_address* or use a value of 0, the Memory Debugger displays all contents beginning at *start_address* and going to the end of the hoard.

The CLI displays hoarded blocks in the order in which your program deallocated them.

–**set** [**on** | **off**]
Enables and disables hoarding.

–**reset** Resets the Memory Debugger settings for hoarding back to their initial value.

–**set_all_deallocs** [**on** | **off**]
Tells the Memory Debugger whether to hoard deallocated blocks.

–**reset_all_deallocs**
Resets the Memory Debugger settings for hoarding of deallocated blocks to its initial value.

–**set_max_kb** *num_kb*
Sets the maximum size of the hoarded information.

–**set_max_blocks** *num_blocks*
Set the maximum number of hoarded blocks.

–**reset_max_kb**
–**reset_max_blocks**
Resets a hoarding size value back to its default.

–**info** [*subcommand*]
Displays information about the heap or regions of the heap within a range of addresses. If you do not use the address arguments, the CLI displays information about all heap allocations.

The information that the Memory Debugger displays includes the start address, a block's length, and other information such as flags or attributes.

–**backtrace**
Tells the CLI to display backtrace information. This list can be very long.

start_address
If you just type a *start_address*, the CLI reports on all allocations beginning at and following this address. If you also type an *end_address*, the CLI limits the display to those allocations between the *start_address* and the *end_address*.

end_address
If you also specify an *end_address*, the CLI reports on all allocations between *start_address* and *end_address*. If you

type 0, it's the same as omitting this argument; that is, the Memory Debugger displays information from the *start_address* to the end of the address space.

–is_dangling *address*
: Indicates if an *address* that was once allocated and not yet recycled by the heap manager is now deallocated.

–leaks
: Locates all memory blocks that your program allocated and which are no longer referenced. That is, using this command tells the Memory Debugger to locate all dangling memory. For more information, see "*Detecting Leaks: dheap –leaks*" on page 57.

  By default, the Memory Debugger only checks to see if the starting location of an allocated memory block is referenced.

  –check_interior
  : Tells the Memory Debugger to consider a memory block as being referenced if the interior portion of it is referenced.

–[no]notify
: Using the –**notify** option tells TotalView to stop your program's execution when the Memory Debugger detects a notifiable event, and then print a message (or display a dialog box if you are also using the GUI) that explains what just occurred. The Memory Debugger can notify you when heap memory errors occur or when tagged blocks are deallocated or reallocated.

  Using the –**nonotify** option tells TotalView not to stop execution. Even if you type the –**nonotify** option, TotalView tracks heap events.

–paint [*subcommands*]
: Enables and disables block painting and shows status information. (For more information on block painting, see "*Block Painting: dheap –paint*" on page 58.)

  [–status]
  : Shows the current paint settings. These are either the values you set using other painting options or their default values.

    If you do not use a subcommand to the –**paint** option, the Memory Debugger shows the block painting status information.

  –set_alloc [on | off]
  –set_dealloc [on | off]
  –set_zalloc [on | off]
  : The *on* options enable block painting. They tell the Memory Debugger to paint a block when your program's heap manager allocates, deallocates, or uses a memory function that sets memory blocks to zero.

    You can only paint zero-allocated blocks if you are also painting regular allocations.

    The *off* options disable block painting.

–reset_alloc
–reset_dealloc
–reset_zalloc

Reset the Memory Debugger settings for block painting to their initial values or to values typed in a startup file.

–set_alloc_pattern *pattern*
–set_dealloc_pattern *pattern*

Set the pattern that the Memory Debugger uses the next time it paints a block of memory. The maximum width of *pattern* can differ between operating systems. However, your pattern can be shorter.

–reset_alloc_pattern
–reset_dealloc_pattern

Reset the patterns used when the Memory Debugger paints memory to the Memory Debugger default values.

–tag_alloc *subcommand* [*start_address* [*end_address*]]

Tells the Memory Debugger to mark a block so that it can notify you when your program deallocates or reallocates a memory block. (For more information, see "*Deallocation Notification: dheap –tag_alloc*" on page 59.)

When tagging memory, if you do not specify address arguments, the Memory Debugger either tags all allocated blocks or removes the tag from all tagged blocks.

–[no]hoard_on_dealloc

Tells the Memory Debugger that it should not release tagged memory back to your program's heap manager for reuse when it is deallocated—this is used in conjunction with hoarding. To reenable memory reuse, use the –nohoard_on_dealloc subcommand. See "*Memory Reuse: dheap –hoard*" on page 55 for more information.

If you use this option, the memory tracker only hoards tagged blocks. In contrast, if you use the **dheap –hoard –set_all_deallocs on** command, the Memory Debugger hoards all deallocated blocks.

–[no]notify_dealloc
–[no]notify_realloc

Enable or disable notification when your program deallocates or reallocates a memory block.

start_address

If you only type a *start_address*, the Memory Debugger either tags or removes the tag from the block that contains this address. The action it performs depends on the subcommand you use.

end_address

If you also specify an *end_address*, the Memory Debugger either tags all blocks beginning with the block containing the *start_address* and ending with the block containing the *end_address* or removes the tag. The action it

	performs depends on the subcommand you use. If *end_address* is 0 (zero) or you do not type an *end_address*, the Memory Debugger tags or removes the tag from all addresses beginning with *start_address* to the end of the heap.
–version	Displays the Memory Debugger version number.

Description: The **dheap** command controls the TotalView Memory Debugger. The Memory Debugger can:

- Tell TotalView to use the Memory Debugger agent to track memory errors.
- Stop execution when a **free()** error occurs, and display information you need to analyze the error. For more information, see "*Notification When free Problems Occur*" on page 54.
- Hoard freed memory so that it is not released to the heap manager. For more information, see "*Memory Reuse: dheap –hoard*" on page 55.
- Detect leaked memory by analyzing if a memory block is reachable. For more information, see "*Detecting Leaks: dheap –leaks*" on page 57.
- Paint memory with a bit pattern when it is allocated and deallocated. For more information, see "*Block Painting: dheap –paint*" on page 58.
- Notify you when a memory block is deallocated or reallocated. For more information, see "*Deallocation Notification: dheap –tag_alloc*" on page 59.

The first step when debugging memory problems is to type the **dheap –enable** command. This command activates the Memory Debugger. You must do this before your program begins executing. If you try to do this after execution starts, TotalView tells you that it will enable the Memory Debugger when you restart your program. For example:

```
d1.<> n
  64 >   int                 num_reds      = 15;
d1.<> dheap -enable
process 1 (30100): This will only take effect on restart
```

You can tell the Memory Debugger to stop execution if:

- A **free()** problem exists by using the **dheap –notify** command.
- A block is deallocated by using the **dheap –tag_alloc –notify_dealloc** command.
- A block is reallocated by using the **dheap –tag_alloc –notify_realloc** command.

If you enable notification, TotalView stops the process when it detects one of these events. The Memory Debugger is always monitoring heap events, even if you turned notification off. That is, disabling notification means that TotalView does not stop a program when events occur. In addition, it does not tell you that the event occurred.

While you can separately enable and disable notification in any group, process, or thread, you probably only want to activate notification on the control group's master process. Because this is the only process that TotalView creates, it is the only process where TotalView can control the Memory

Debugger's environment variable. For example, slave processes are normally created by an MPI starter process or as a result of using the **fork()** and **exec()** functions. In these cases, TotalView simply attaches to them. For more information, see Chapter 4, "*Creating Programs for Memory Debugging*," on page 61.

If you do not use a **dheap** subcommand, the CLI displays memory status information. You only use the **–status** option when you want the CLI to display status information in addition to doing something else.

The information that the **dheap** command displays can contain a flag containing additional information about the memory location. The following table describes these flags:

Flag Value	Meaning
0x0001	Operation in progress
0x0002	**notify_dealloc**: you will be notified if the block is deallocated
0x0004	**notify_realloc**: you will be notified if the block is reallocated
0x0008	**paint_on_dealloc**: the Memory Debugger will paint the block when it is deallocated
0x0010	**dont_free_on_dealloc**: the Memory Debugger will not free the tagged block when it is deallocated
0x0020	**hoarded**: the Memory Debugger is hoarding the block

While some **dheap** options obtain information on specific memory conditions, you can use the following options throughout your session:

- **dheap** or **dheap –status**: Displays Memory Debugger state information. For example:

```
a1.<> dheap -status
              process:    Enable   Notify    Available
    1         (18868):       yes      yes          yes
    2         (18947):       n/a      yes          yes
    3         (18953):       n/a      yes          yes
    4         (18956):       n/a      yes          yes
```

- **dheap –version**: Displays version information. You receive information for each process as processes can be compiled with different versions of the Memory Debugger. For example:

```
a1.<> dheap -version
              process:      Version
    1         (18868):        1.001
    2         (18947):        1.001
    3         (18953):        1.001
    4         (18956):        1.001
```

- **dheap –backtrace**: Displays information about how much of the backtrace is being displayed. For example:

```
a1.<> dheap -backtrace
              process:     Depth      Trim
    1         (18868):        32         5
    2         (18947):        32         5
    3         (18953):        32         5
    4         (18956):        32         5
```

Using arguments to this command, you can change both the *depth* and the *trim* values. Changing the *depth* value changes the number of stack frames that the Memory Debugger displays in a backtrace display. Changing the *trim* value eliminates the topmost stack frames.

- **dheap –info**: Displays information about currently allocated memory blocks. For example:

```
d1.<> dheap -info
process     1    (5320):
         0x8049790 --       0x804979a        0xa [       10]
  flags: 0x0 (none)
         0x80497a0 --       0x80497b4       0x14 [       20]
  flags: 0x0 (none)
         0x80497b8 --       0x80497d6       0x1e [       30]
  flags: 0x0 (none)
         0x80497e0 --       0x8049808       0x28 [       40]
  flags: 0x0 (none)
```

Notification When free Problems Occur

If you type **dheap –enable –notify** and then run your program, the Memory Debugger notifies you if a problem occurs when your program tries to free memory. (For more information, see Chapter 15 of the *TotalView Users Guide*.)

When execution stops, you can type **dheap** (with no arguments), to show information about what happened. You can also use the **dheap –info** and **dheap –info –backtrace** commands to display additional information. The information displayed by these commands lets you locate the statement in your program that caused the problem. For example:

```
d1.<> dheap
         process:   Enable   Notify    Available
 1       (18993):      yes      yes          yes
 1.1  realloc: Address does not match any allocated block.:
            0xbfffd87c
```

For each allocated region, the CLI displays the start and end address, and the length of the region in decimal and hexadecimal formats. For example:

```
d1.<> dheap
           process:   Enable   Notify   Available
 1         (30420):      yes      yes         yes
 1.1  free: Address is not the start of any allocated block.:
      free:  existing allocated block:
      free:  start=0x08049b00  length=(17 [0x11])
      free:  flags: 0x0 (none)
      free:   malloc       PC=0x40021739 [/.../
                  malloc_wrappers_dlopen.c]
      free:   main         PC=0x0804871b [../free_prob.c]
      free:   __libc_start_main PC=0x40140647 [/lib/i686/
                  libc.so.6]
      free:   _start       PC=0x080485e1 [/.../free_prob]

      free:  address passed to heap manager: 0x08049b08
```

The Memory Debugger can also tell you when tagged blocks are deallocated or reallocated. For more information, see "*Deallocation Notification: dheap –tag_alloc*" on page 59.

Showing Backtrace Information: dheap –backtrace:

The backtrace associated with a memory allocation can contain many stack frames that are part of the heap library, the Memory Debugger's library, and other related functions and libraries. You are not usually interested in this information, since these stack frames aren't part of your program. Using the **–backtrace** option lets you manage this information, as follows:

- **dheap –backtrace –set_trim** *value*

  Tells the Memory Debugger to remove—that is, trim—this number of stack frames from the top of the backtrace. This lets you *hide* the stack frames that you're not interested in as they come from libraries.

- **dheap –backtrace –set_depth** *value*

  Tells the Memory Debugger to limit the number of stack frames to the value that you type as an argument. The *depth* value starts after the *trim* value. That is, the number of excluded frames does not include the frames that were trimmed.

Memory Reuse: dheap –hoard

In some cases, you may not want your system's heap manager to immediately reuse memory. You would do this, for example, when you are trying to find problems that occur when more than one process or thread is allocating the same memory block. Hoarding allows you to temporarily delay the block's release to the heap manager. When the hoard has reached its capacity in either size or number of blocks, the Memory Debugger releases previously hoarded blocks back to your program's heap manager.

The order in which the Memory Debugger releases blocks is the order in which it hoards them. That is, the first blocks hoarded are the first blocks released—this is a first-in, first-out (fifo) queue.

Hoarding is a two-step process, as follows:

1. Use the **dheap –enable** command to tell the Memory Debugger to track heap allocations.
2. Use the **dheap –hoard –set on** command to tell the Memory Debugger not to release deallocated blocks back to the heap manager. (The **dheap –hoard –set off** command tells the Memory Debugger to no longer hoard memory.) After you turn hoarding on, use the **dheap –hoard –set_all_deallocs on** command to tell the Memory Debugger to start hoarding blocks.

At any time, you can obtain the hoard's status by typing the **dheap –hoard** command. For example:

```
d1.<> dheap -hoard
                             All      Max     Max
 process: Enabled  deallocs     size  blocks      Size    Blocks
1 (10883):     yes        yes  16 (kb)      32  15 (kb)        9
```

The **Enabled** column contains either **yes** or **no**, which indicates whether hoarding is enabled. The **All deallocs** column indicates if hoarding is occur-

3. Using dheap

ing. The next columns show the maximum size in kilobytes and number of blocks to which the hoard can grow. The last two columns show the current size of the hoard, again, in kilobytes and the number of blocks.

As your program executes, the Memory Debugger adds the deallocated region to a FIFO buffer. Depending on your program's use of the heap, the hoard could become quite large. You can control the hoard's size by setting the maximum amount of memory in kilobytes that the Memory Debugger can hoard and the maximum number of hoarded blocks.

dheap –hoard –set_max_kb *num_kb*
: Sets the maximum size in kilobytes to which the hoard is allowed to grow. The default value on many operating systems is 32KB.

dheap –hoard –set_max_blocks *num_blocks*
: Sets the maximum number of blocks that the hoard can contain.

You can tell which blocks are in the hoard by typing the **dheap –hoard –display** command. For example:

```
d1.<> dheap -hoard -display
process     1   (10883):
              0x804cdb0 --        0x804d3b0      0x600 [     1536]
    flags: 0x32 (hoarded)
              0x804d3b8 --        0x804dab8      0x700 [     1792]
    flags: 0x32 (hoarded)
              0x804dac0 --        0x804e2c0      0x800 [     2048]
    flags: 0x32 (hoarded)
              0x804fce8 --        0x804fee8      0x200 [      512]
    flags: 0x32 (hoarded)
              0x804fef0 --        0x80502f0      0x400 [     1024]
    flags: 0x32 (hoarded)
```

Checking for Dangling Pointers: dheap –is_dangling:

The **dheap –is_dangling** command lets you determine if a pointer is still pointing into a deallocated memory block.

You can also use the **dheap –is_dangling** command to determine if an address refers to a block that was once allocated but has not yet been recycled. That is, this command lets you know if a pointer is pointing into deallocated memory.

Here's a small program that illustrates a dangling pointer:

```
main(int argc, char **argv)
{
    int *addr = 0;          /* Pointer to start of block.     */
    int *misaddr = 0;       /* Pointer to interior of block.  */

    addr = (int *) malloc (10 * sizeof(int));
                           /* Point to interior of the block. */
    misaddr = addr + 5;
```

```
                                /* addr and misaddr now dangling.  */
      free (addr);
      printf ("addr=%lx, misaddr=%lx\n",
                (long) addr, (long) misaddr);
  }
```

If you set a breakpoint on the **printf()** statement and probe the addresses of **addr** and **misaddr**, the CLI displays the following:

```
d1.<> dheap -is_dangling 0x80496d0
              process:          0x80496d0
    1         (19405):           dangling

d1.<> dheap -is_dangling 0x80496e4
              process:          0x80496e4
    1         (19405):  dangling interior
```

This example is contrived. When creating this example, the variables were examined for their address and their addresses were used as arguments. In a realistic program, you'd find the memory block referenced by a pointer and then use that value. In this case, because it is so simple, using the CLI **dprint** command gives you the information you need. For example:

```
d1.<> dprint addr
 addr = 0x080496d0 (Dangling) -> 0x00000000 (0)
d1.<> dprint misaddr
 misaddr = 0x080496e4 (Dangling Interior) -> 0x00000000 (0)
```

If a pointer is pointing into memory that is deallocated, and this memory is being hoarded, the CLI also lets you know that you are looking at hoarded memory.

Detecting Leaks: dheap –leaks

The **dheap –leaks** command locates memory blocks that were allocated and are no longer referenced. It then displays a report that describes these blocks; for example:

```
d1.<> dheap -leaks
process  1   (32188):  total count 9, total bytes 450
 * leak  1 -- total count 9 (100.00%), total bytes 450 (100%)
         -- smallest / largest / average leak:  10 / 90 / 50
    :  malloc   PC=0x40021739 [/.../malloc_wrappers_dlopn.c]
    :  main     PC=0x0804851e [/.../local_leak.cxx]
    :  __libc_start_main PC=0x40055647 [/lib/i686/libc.so.6]
    :  _start   PC=0x080483f1 [/.../local_leak]
```

If you use the –**check_interior** option, the Memory Debugger considers a block as being referenced if a pointer exists to memory inside the block.

In addition to providing backtrace information, the CLI:

- Consolidates leaks made by one program statement into one leak report. For example, leak 1 has nine instances.
- Reports the amount of memory consumed for a group of leaks. It also tells you what percentage of leaked memory this one group of memory is using.
- Indicates the smallest and largest leak size, as well as telling you what the average leak size is for a group.

You might want to paint a memory block when it is deallocated so that you can recognize that the data pointed to is out-of-date. Tagging the block so that you can be notified when it is deallocated is another way to locate the source of problems.

Block Painting: dheap –paint

When your program allocates or deallocates a block, the Memory Debugger can paint the block with a bit pattern. This makes it easy to identify uninitialized blocks, or blocks pointed to by dangling pointers.

Here are the commands that enable block painting:

- dheap –paint –set_alloc on
- dheap –paint –set_dealloc on
- dheap –paint –set_zalloc on

Use the **dheap –paint** command to check the kind of painting that occurs and what the current painting pattern is. For example:

```
d1.<> dheap -paint
                                             Alloc      Dealloc
   process: Alloc Dealloc AllocZero        pattern      pattern
1   (1012):   yes     yes        no     0xa110ca7f   0xdea110cf
```

Some heap allocation routines such as **calloc()** return memory initialized to zero. Using the –**set_zalloc_on** command allows you to separately enable the painting of the memory blocks altered by these kinds of routines. If you do enable painting for routines that set memory to zero, the Memory Debugger uses the same pattern that it uses for a normal allocation.

Here's an example of painted memory:

```
d1.<> dprint *(red_balls)
 *(red_balls) = {
   value = 0xa110ca7f (-1592735105)
   x = -2.05181867705792e-149
   y = -2.05181867705792e-149
   spare = 0xa110ca7f (-1592735105)
   colour = 0xa110ca7f -> <Bad address: 0xa110ca7f>
 }
```

The **0xall0ca7f** allocation pattern resembles the word "allocate". Similarly, the **0xdea110cf** deallocation pattern resembles "deallocate".

Notice that all of the values in the **red_balls** structure in this example aren't set to **0xall0ca7f**. This is because the amount of memory used by elements of the variable use more bits than the **0xall0ca7f** bit pattern. The following two CLI statements show the result of printing the **x** variable, and then casting it into an array of two integers:

```
d1.<>  dprint (red_balls)->x
 (red_balls)->x = -2.05181867705792e-149
d1.<> dprint {*(int[2]*)&(red_balls)->x}
 *(int[2]*)&(red_balls)->x = {
   [0] = 0xa110ca7f (-1592735105)
   [1] = 0xa110ca7f (-1592735105)
```

(Diving in the GUI is much easier.)

You can tell the Memory Debugger to use a different pattern by using the following two commands:

- **dheap –paint –set_alloc_pattern** *pattern*
- **dheap –paint –set_dealloc_pattern** *pattern*

Deallocation Notification: dheap –tag_alloc

You can tell the Memory Debugger to tag information within the Memory Debugger's tables and to notify you when your program either frees a block or passes it to **realloc()** by using the following two commands:

- **dheap –tag_alloc –notify_dealloc**
- **dheap –tag_alloc –notify_realloc**

Tagging is done within the Memory Debugger's agent. It tells the Memory Debugger to watch those memory blocks. Arguments to these commands tell the Memory Debugger which blocks to tag. If you do not type address arguments, TotalView notifies you when your program frees or reallocates an allocated block. The following example shows how to tag a block and how to see that a block is tagged:

```
d1.<> dheap -tag_alloc -notify_dealloc 0x8049a48
process     1   (19387):  1 record(s) update
d1.<> dheap -info
process     1   (19387):
         0x8049a48 --        0x8049b48      0x100 [      256]
  flags: 0x2 (notify_dealloc)
         0x8049b50 --        0x8049d50      0x200 [      512]
  flags: 0x0 (none)
         0x8049d58 --        0x804a058      0x300 [      768]
  flags: 0x0 (none)
```

Using the –**notify_dealloc** subcommand tells the Memory Debugger to let you know when a memory block is freed or when **realloc()** is called with its length set to zero. If you want notification when other values are passed to the **realloc()** function, use the –**notify_realloc** subcommand.

After execution stops, here is what the CLI displays when you type another **dheap –info** command:

```
d1.<> dheap -info
process     1   (19387):
         0x8049a48 --        0x8049b48      0x100 [      256]
  flags: 0x3 (notify_dealloc, op_in_progress)
         0x8049b50 --        0x8049d50      0x200 [      512]
  flags: 0x0 (none)
         0x8049d58 --        0x804a058      0x300 [      768]
```

TV_HEAP_ARGS

Environment variable for presetting Memory Debugger values

When you start TotalView, it looks for the **TV_HEAP_ARGS** environment variable. If it exists, TotalView reads values placed in it. If one of these values changes a Memory Debugger default value, the Memory Debugger uses this value as the default.

If you select a **<Default>** button in the GUI or a reset option in the CLI, the Memory Debugger resets the value to the one you set here, rather than to its default.

TV_HEAP_ARGS Values

The values that you can enter into this variable are as follows:

display_allocations_on_exit
Tells the Memory Debugger to dump the allocation table when your program exits. If your program ends because it received a signal, the Memory Debugger might not be able to dump this table.

backtrace_depth *depth*
Sets the backtrace depth value. See "*Showing Backtrace Information: dheap –backtrace:*" on page 55 for more information.

backtrace_trim *trim*
Sets the backtrace trim value. See "*Showing Backtrace Information: dheap –backtrace:*" on page 55 for more information.

memalign_strict_alignment_even_multiple
The Memory Debugger provides an integral multiple of the alignment rather than the even multiple described in the Sun **memalign** documentation. By including this value, you are telling the Memory Debugger to use the Sun alignment definition. However, your results might be inconsistent if you do this.

output fd *int*
output file *pathname*
Sends output from the Memory Debugger to the file descriptor or file that you name.

verbosity *int*
Sets the Memory Debugger's verbosity level. If the level is greater than 0, the Memory Debugger sends information to **stderr**. The values you can set are:

0: Display no information. This is the default.

1: Print error messages.

2: Print all relevant information.

This option is most often used when debugging Memory Debugger problems. Setting the TotalView **VERBOSE** CLI variable does about the same thing.

Example: When you are entering more than one value, separate entries with spaces. For example:

```
setenv TV_HEAP_ARGS output file “my_file backtrace_depth 16”
```

Creating Programs for Memory Debugging

4

The TotalView Memory Debugger puts its heap agent between your program and its heap library. This allows the agent to intercept the calls that your program makes to this library. After it intercepts the call, it checks it for errors, and then sends it on to the library so that it can be processed. The Memory Debugger agent does not replace standard memory functions; it just monitors what they do. For more information, see "*Behind the Scenes*" on page 5.

You can incorporate the agent into your environment either by:

- Linking your application with the agent.
- Requesting that the agent's library be preloaded by setting a run-time loader environment variable. This is only done when your program will attach to another program that it did not start and you want the Memory Debugger to locate problems in this second application.

AIX applications differ from applications running on other platforms as AIX does not support interposition. However, TotalView can replace the AIX heap library.

Topics in this section are:

- "*Linking Your Application With the Agent*" on page 61
- "*Attaching to Programs*" on page 63

Linking Your Application With the Agent

In some situations, you need to explicitly link the Memory Debugger's agent directly to your program. For example, if you are debugging an MPI program, your starter program might not propagate environment variables.

On AIX, you must always link your program so that **malloc()** *can find the heap replacement and agent. In addition, you only set your* **LIBPATH** *environment variable when the* **tvheap_mr.a** *library is in your* **LIBPATH**. *If it isn't, your program might not load. You must use the* **–L** *options listed in the following table.*

The following table lists additional linker command-line options that you must use when you link your program:

Platform	Compiler	ABI	Additional linker options
HP Tru64 Alpha (version 5)	Compaq/KCC	64	–L*path* –ltvheap –rpath *path*
	GCC	64	–L*path* –ltvheap –Wl,–rpath,*path*
IBM RS/6000 (all)	IBM/GCC	32/64	–L*path_mr* –L*path*
	KCC	32	–L*path_mr* –L*path* --static_libKCC
		64	–L*path_mr* –L*path*
AIX 4	IBM/KCC	32	–L*path_mr* –L*path* *path*/aix_malloctype.so \ –binitfini:aix_malloctype_init
		64	–L*path_mr* –L*path* *path*/aix_malloctype64_4.so \ –binitfini:aix_malloctype_init
	GCC	32	–L*path_mr* –L*path* \ *path*/aix_malloctype.so –Wl, –binitfini:aix_malloctype_init
		64	–L*path_mr* –L*path* \ *path*/aix_malloctype64_4.so –Wl, –binitfini:aix_malloctype_init
AIX 5	IBM/GCC/KCC	32	–L*path_mr* –L*path* *path*/aix_malloctype.o
		64	–L*path_mr* –L*path* *path*/aix_malloctype64_5.o
Linux x86	GCC/Intel/PGI	32	–L*path* –ltvheap –Wl,–rpath,*path*
	KCC	32	–L*path* –ltvheap –rpath *path*
Linux x86-64	GCC/PGI	32	–L*path* –ltvheap –Wl,–rpath,*path*
		64	–L*path* –ltvheap_64 –Wl,–rpath,*path*
Linux IA64	GCC/Intel	64	–L*path* –ltvheap –Wl,–rpath,*path*
SGI	SGI/GCC/KCC	32	–L*path* –ltvheap –rpath *path*
		64	–L*path* –ltvheap_64 –rpath *path*
Sun	Sun/KCC/ Apogee	32	–L*path* –ltvheap –R *path*
	Sun/KCC	64	–L*path* –ltvheap_64 –R *path*
	GCC	32	–L*path* –ltvheap –Wl,–R,*path*
		64	–L*path* –ltvheap_64 –Wl,–R,*path*

The following list describes the options in this table:

path The absolute path to the agent in the TotalView installation hierarchy. More precisely, this directory is:

installdir/toolworks/totalview.*version*/*platform*/lib

installdir The installation base directory name.

version The TotalView version number.

platform The platform tag.

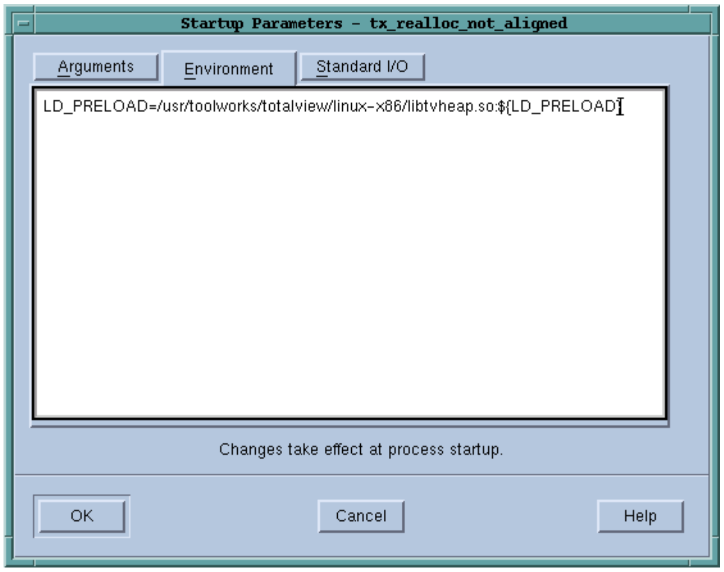
path_mr The absolute path of the heap replacement library. This value is determined by the person who installs the TotalView malloc replacement library.

Since it is easy to misinterpret the path specifications, you may want to see what value TotalView uses when it sets a path. Here's the procedure:

1 Start TotalView.

2 Enable the Memory Debugger by selecting the **Tools > Memory Debugger** command, and then checking the **Enable memory debugging** checkbox.

3 Select the **Process** > **Startup Parameters** command and then select the **Environment** Page. Type a value that is the same as or similar to the one in the following figure:



```
LD_PRELOAD=/usr/toolworks/totalview/linux-x86/libtvheap.so:${LD_PRELOAD}
```

Changes take effect at process startup.

Attaching to Programs

When your program attaches to a process that is already running, the Memory Debugger can not locate heap problems in that process unless you manually set a Memory Debugger environment variable. The variable that you use must be unique (or relatively so) on each platform. The following table lists these variables:

Platform	Variable
HP Tru64 Alpha	_RLD_LIST
IBM AIX	MALLOCTYPE
Linux IA64 and x86	LD_PRELOAD
SGI Irix	__RLDN32_LIST _RLD64_LIST
Sun	LD_PRELOAD

You can display the value that TotalView uses by displaying the **Environment** Page within the **Process** > **Startup Parameters** command. To set this variable:

1 Start TotalView and enable memory debugging.
2 Open this dialog box and see what the value is for your environment.
3 Close TotalView.
4 Start the program to which you will be attaching as an argument to the **env** command. For example, here's how to set this variable on AIX:

```
env MALLOCTYPE user:tvheap_mr.a totalview my_prog
```

Do not set these environment variables so that the agent interposes itself when you execute any command. For example, use **env** *to set this variable and run TotalView rather than* **setenv**. *If you use* **setenv**, *you will run the agent against all your programs, including system programs such as* **ls**.

Using the Memory Debugger

This section describes using the Memory Debugger in various environments. This section describes the following environments and platforms:

- MPICH
- IBM PE
- SGI MPI
- RMS MPI

MPICH

You use the Memory Debugger with MPICH MPI codes as follows. (Etnus has tested this only on Linux x86.)

1. You must link your parallel application with the Memory Debugger's agent, as described in "*Linking Your Application With the Agent*" on page 61. On most Linux x86 systems, type:

   ```
   mpicc -g test.o -o test -Lpath -ltvheap -Wl,-rpath,path
   ```

2. Start TotalView using the **–tv** command-line option to the **mpirun** script in the usual way; for example:

   ```
   mpirun -tv mpirun-args test args
   ```

   TotalView starts up on the rank 0 process.

   Because you linked in the Memory Debugger's agent, memory debugging is automatically selected in your rank 0 process.
3. If you need to, configure the Memory Debugger.
4. Run the rank 0 process.

IBM PE

You can use the Memory Debugger with IBM PE MPI codes.There are two alternatives.

You will not be able to install tvheap_mr.a under AIX *on your target system unless you have installed the bos.adt.syscalls package, which is part of the System Calls Application Development Toolkit.*

The first is to place the following **proc** in your **.tvdrc** file:

```
# Automatically enable memory error notifications
# (without enabling memory debugging) for poe programs.
proc enable_mem {loaded_id} {
   set mem_prog poe
   set executable_name [TV::image get $loaded_id name]
   set file_component [file tail $executable_name]

   if {[string compare $file_component $mem_prog] == 0} {
      puts "Enabling Memory Debugger for $file_component”
      dheap -notify
   }
}

# Append this proc to the TotalView image load callbacks
# so that it runs this macro automatically.
dlappend TV::image_load_callbacks enable_mem
```

Here's the second method:

1 You must prepare your parallel application to use the Memory Debugger's agent, as described in "*Linking Your Application With the Agent*" on page 61 and "*Installing tvheap_mr.a on* AIX" on page 66. Here is an example that usually works:

```
mpcc_r -g test.o -o test -Lpath_mr -Lpath \
      path/aix_malloctype.o
```

"*Installing tvheap_mr.a on* AIX" on page 66 contains additional information.

2 Start TotalView on **poe** in the usual way:

```
totalview poe -a test args
```

Because **tvheap_mr.a** *is not in* **poe**'*s* LIBPATH, *enabling the Memory Debugger on the* **poe** *process causes problems because* **poe** *cannot locate the* **tvheap_mr.a** *malloc replacement library.*

3 If you want TotalView to notify you when a heap error occurs in your application (and you probably do), use the CLI to turn on notification, as follows:

- ➤ Open a CLI window by selecting the **Tools** > **Command Line** command from the Process Window showing **poe**.
- ➤ In a CLI window, enter the **dheap –notify** command. This command turns on notification in the **poe** process. The MPI processes to which TotalView attaches inherit notification.

4 Run the **poe** process.

SGI MPI

There are two ways to use the Memory Debugger on SGI MPI code. In most cases, all you need do is select the **Tools** > **Memory Debugging** command, select the **mpirun** process in the **Process Set** area, and then check the **Enable memory debugging** check box on the **mpirun** process. Occasionally, this can cause a problem. If it does, here's what you should do:

1 Link your parallel application with the Memory Debugger's agent, as described in the *Debugging Memory Problems* chapter of the *TotalView Users Guide*. Basically, the command you will enter is:

```
cc -n32 -g test.o -Lpath -ltvheap -rpath path \
      -lmpi -o test
```

2 Start TotalView on the **mpirun** process. For example:

```
totalview mpirun -a mpirun-args test args
```

3 If you need to, configure the Memory Debugger.

4 Run the **mpirun** process.

RMS MPI

Here's how to use the Memory Debugger with Quadrics RMS MPI codes. (Etnus has tested this only on Linux x86.)

1 You do not need to link the application with the Memory Debugger because the **prun** process propagates environment variables to the rank processes. However, if you'd like to link the application with the Memory Debugger's agent, you can.

2 Start TotalView on **prun**; for example:

```
totalview prun -a prun-args test args
```

3 Enable memory debugging by selecting the **Tools > Memory Debugging** command, selecting the **mpirun** process in the **Process Set** area, and then checking the **Enable memory debugging** check box. If you had linked in the agent, this option is automatically selected.
4 If you want TotalView to notify you when a heap error occurs in your application (and you probably do), check the **Stop execution when an allocation or deallocaction error occurs** check box.
5 Run the **prun** process.

Installing tvheap_mr.a on AIX

You must install the **tvheap_mr.a** library on each node upon which you will be running the Memory Debugger agent. One way to do this is to place a symbolic link in **/usr/lib** that points to the **tvheap_mr.a** library. If you do this, you do not need to add special **–L** command-line options to your build. In addition, there are no special requirements when using **poe**.

The rest of this section describes what you need to do if you cannot create symbolic links. Even when you create symbolic links, you will still need to recreate **tvheap_mr.a** whenever **libc.a** changes.

The **aix_install_ tvheap_mr.sh** script contains most of what you need to do. This script is in the following directory:

  toolworks/totalview.version/rs6000/lib/

For example, after you become root, enter the following commands:

```
cd toolworks/totalview.6.3.0-0/rs6000/lib
mkdir /usr/local/tvheap_mr
./aix_install_tvheap_mr.sh ./tvheap_mr.tar /usr/local/tvheap_mr
```

Use **poe** to create **tvheap_mr.a** on multiple nodes.

The pathname for the **tvheap_mr.a** library must be the same on each node. This means that you cannot install this library on a shared file system. Instead, you must install it on a file system that is private to the node. For example, because **/usr/local** is usually only accessible from the node upon which it is installed, you might want to install it there.

The **tvheap_mr.a** library depends heavily on the exact version of **libc.a** that is installed on a node. If **libc.a** changes, you must recreate **tvheap_mr.a** by re-executing the **aix_install_tvheap_mr.sh** script.

LIBPATH and Linking

This section discusses compiling and linking your AIX programs. The following command adds *path_mr* and *path* to your program's default **LIBPATH**:

```
xlc -Lpath_mr -Lpath -o a.out foo.o
```

When **malloc()** dynamically loads **tvheap_mr.a**, it should find the library in *path_mr*. When **tvheap_mr.a** dynamically loads **tvheap.a**, it should find it in *path*.

The AIX linker allows you to relink executables. This means that you can make an already complete application ready for the Memory Debugger's agent; for example:

```
cc a.out -Lpath_mr -Lpath -o a.out.new
```

Here's an example that does not link in the heap replacement library. Instead, it allows you to dynamically set **MALLOCTYPE**:

```
xlC -q32 -g \
   -L/usr/local/tvheap_mr \
   -L/home/totalview/interposition/lib prog.o -o prog
```

The next example shows how you allow your program to access the Memory Debugger's agent by linking in the **aix_malloctype.o** module:

```
xlc -q32 -g \
   -L/usr/local/tvheap_mr \
   -L/home/totalview/interposition/lib prog.o \
     /home/totalview/interposition/lib/aix_malloctype.o \
   -o prog
```

You can check that the paths made it into the executable by running the **dump** command; for example:

```
% dump -Xany -Hv tx_memdebug_hello

   tx_memdebug_hello:

           ***Loader Section***
         Loader Header Information
  VERSION#      #SYMtableENT   #RELOCent     LENidSTR
  0x00000001    0x0000001f     0x00000040    0x000000d3

  #IMPfilID     OFFidSTR       LENstrTBL     OFFstrTBL
  0x00000005    0x00000608     0x00000080    0x000006db

           ***Import File Strings***
  INDEX  PATH                BASE          MEMBER
  0      /.../interpos/lib:/usr/.../lib:/usr/lib:/lib
  1                          libc.a            shr.o
  2                          libC.a            shr.o
  3                          libpthreads.a     shr_comm.o
  4                          libpthreads.a     shr_xpg5.o
```

Index 0 in the **Import File Strings** section shows that the search path the runtime loader uses when it dynamically loads a library. Some MPI systems propagate the preload library environment to the processes they will run; others, do not. If they do not, you need to manually link them with the **tvheap** library.

In some circumstances, you might want to link your program instead of setting the **MALLOCTYPE** environment variable. If you set the **MALLOCTYPE** environment variable for your program and it fork/execs a program that is not linked with the agent, your program will terminate because it fails to find **malloc()**.

Index